NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SIGNAL DETECTION AND FRAME SYNCHRONIZATION OF MULTIPLE WIRELESS NETWORKING WAVEFORMS

by

Keith C. Howland

September 2007

Thesis Advisor:	Murali Tummala
Co-Advisor:	John McEachen

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2007	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Signal Detection and Frame Synchronization of Multiple Wireless Networking Waveforms	5. FUNDING NUMBERS
6. AUTHOR(S) Keith C. Howland	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

This thesis investigates the detection, classification, frame synchronization, and demodulation of wireless networking waveforms by a digital receiver. The approach is to develop detection thresholds for wireless networking signals based upon the probability density functions of the signal present or signal absent scenarios. A Neyman-Pearson test is applied to determine decision thresholds and the associated probabilities of detection. With a chosen threshold, MATLAB simulations are run utilizing models developed to generate and receive IEEE 802.11a, IEEE 802.16, and IEEE 802.11b signals in multipath channels characterized by Rayleigh fading. Algorithms are developed for frame synchronization for each of the three waveforms. The probability of signal detection, successful frame synchronization, and the bit error rates of the received packet header and data are calculated. The results show that, even in Rayleigh fading environments at low signal to noise levels, these three waveforms can be distinguished in a digital receiver. Further, the results show that significant signal information can be gathered on these wireless networking waveforms, even when the entire signal cannot be demodulated due to low signal to noise ratios.

14. SUBJECT TERMS Signal Detection, Frame Synchronization, Orthogonal Frequency Division Multiplexing (OFDM), IEEE 802.11a, IEEE 802.16	15. NUMBER OF PAGES 191
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SIGNAL DETECTION AND FRAME SYNCHRONIZATION OF MULTIPLE WIRELESS NETWORKING WAVEFORMS

Keith C. Howland
Lieutenant Commander, United States Navy
B.A., Cornell University, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2007

Author: Keith Howland

Approved by: Professor Murali Tummala
Thesis Advisor

Professor John McEachen
Co-Advisor

Professor Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis investigates the detection, classification, frame synchronization, and demodulation of wireless networking waveforms by a digital receiver. The approach is to develop detection thresholds for wireless networking signals based upon the probability density functions of the signal present or signal absent scenarios. A Neyman-Pearson test is applied to determine decision thresholds and the associated probabilities of detection. With a chosen threshold, MATLAB simulations are run utilizing models developed to generate and receive IEEE 802.11a, IEEE 802.16, and IEEE 802.11b signals in multipath channels characterized by Rayleigh fading. Algorithms are developed for frame synchronization for each of the three waveforms. The probability of signal detection, successful frame synchronization, and the bit error rates of the received packet header and data are calculated. The results show that, even in Rayleigh fading environments at low signal to noise levels, these three waveforms can be distinguished in a digital receiver. Further, the results show that significant signal information can be gathered on these wireless networking waveforms, even when the entire signal cannot be demodulated due to low signal to noise ratios.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND/OR ABBREVIATIONS

AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BPSK	Binary Phase-Shift Keying
CP	Cyclic Prefix
CRC	Cyclic Redundancy Check
dB	Decibel
ICI	Interchannel Interference
IDFT	Inverse Discrete Fourier Transform
IEEE	Institute of Electrical and Electronics Engineers
ISI	Intersymbol Interference
MAC	Medium Access Control
MPDU	MAC Protocol Data Unit
OFDM	Orthogonal Frequency Division Multiplexing
PDF	Probability Density Function
PLCP	Physical Layer Convergence Procedure
PPDU	PLCP Packet Data Unit
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase-Shift Keying
ROC	Receiver Operating Characteristic
rms	Root Mean Square
SNR	Signal to Noise Ratio
WLAN	Wireless Local Area Network

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

With the emergence of network-centric warfare, the goal of military wireless communications is to network mobile units together so that data can be shared while using bandwidth efficiently. As the demand grows to network more diverse units, each node requires the ability to communicate on different frequencies with different waveforms. Packet-based wireless networks, utilizing digital communication techniques, have emerged as an effective implementation of the network-centric warfare model. With the rapid deployment of the IEEE 802.11x standards, the commercial sector has already recognized the advantages of packet-based, wireless networking and the need for fast, bandwidth-efficient systems. Furthermore, the commercial sector is also recognizing the demand for a single user to be able to seamlessly switch between the diverse array of networks available and take advantage of the fastest connection. Similarly, a military user will wish to communicate over several different links, e.g., one for command and control data and three for sensor data.

In both the military and commercial markets, the need arises for the receiver to be able to detect and classify different wireless signals. This could be accomplished with a separate receiver for each waveform. This approach would require an ever-growing footprint and would not be practical for the ability to communicate on many different links. A better approach is a single, air interface capable of sampling the different signals at the required sampling rates, and a digital signal processing backend capable of detecting and demodulating the different waveforms.

The objective of this thesis is to investigate signal detection and frame synchronization for wireless network signals. Specifically, a receiver capable of differentiating between and synchronizing to three commercial wireless networking standards - IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 - will be developed. Although the current commercial implementations of these standards operate in separate sections of the frequency spectrum, the intent of this thesis is to investigate their joint signal detection independent of carrier frequency. This implementation would require a

digital receiver with a wideband air interface. These three waveforms were chosen because of their dominance in the commercial wireless networking market and because they offer an excellent example for military wireless networking applications.

Signal detection is challenging in packet-switched, contention-based, wireless networks because data is sent in bursts. Since the receiver does not know when to expect the next frame, each frame must carry extra information to allow for detection and frame synchronization. In the IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 standards, a header section is added at the front of each frame to accomplish this task. The header includes a signal, called the preamble, which is the same for each frame. The preamble is intended to identify the frame as belonging to a certain standard. The receiver, then, must have a way of recognizing the preamble. This thesis describes the properties of each of the preambles of the IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 standards. Following this, an effective preamble recognition technique in which the receiver maintains a copy of each preamble of interest is presented. The receiver compares every received signal with the stored preambles to see if there is a match. This method of comparison is accomplished through the cross-correlation of the received signal with the stored preambles. The preambles were designed so that when two versions of the same preamble are aligned perfectly, the result is approximately an order of magnitude larger than when they are not aligned perfectly or when the preamble is not present. Signal detection occurs when the cross-correlation exceeds a predetermined threshold.

For a digital receiver to detect each of these signals without misidentifying any of them, each of the preambles must have low values when cross-correlated with the others. This thesis investigates the cross-correlation properties of the preambles. Following this, the thesis defines a decision threshold appropriate for joint detection of all three of the above-mentioned wireless networking signals. A decision threshold is chosen using the Neyman-Pearson test. In the Neyman-Pearson test, a decision threshold is determined based on a desired probability of false alarm. The probability of false alarm is the likelihood that the receiver will determine a signal as present when it is not. Once a decision threshold has been found, its associated probability of detection can be

determined. In this way, a decision threshold is found for the receiver that can be applied to each of the signals of interest to generate acceptable probabilities of detection and probabilities of false alarm.

Once the receiver has detected a particular signal, the following task is to achieve frame synchronization. Frame synchronization is important in digital communications because bits are encoded onto sinusoidal waveforms over defined intervals termed bit periods. In order for the receiver to be able to recover the transmitted bit, the receiver must integrate the received waveform over this bit period. If receiver starts the integration period too late or too early, the captured energy will be a mix of two different bits. If the timing offset is significant, a high bit error rate will result, and the message will not be received. Thus, in order to properly demodulate a received signal, the receiver must be able to determine the start of the first bit period of the transmission.

The thesis presents a frame synchronization technique that uses the cross-correlation result to find the first data symbol in the frame. The technique takes advantage of the fact that the place of the preamble within the signal header is known a priori. The technique also takes advantage of the fact that when the output of the cross-correlation rises above the threshold, the last received sample in the filter should be the last sample of the received preamble. In other words, the cross-correlation peak can be used to find which sample corresponds to the end of the preamble. Once this sample is located, the first data sample is always known to be a fixed number of samples away.

Models were created in MATLAB for the generation of IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 frame headers. These baseband signals were passed through a filter simulating a non-line of sight, multipath, fading channel. The resulting signals were then processed by the receiver in parallel as shown in Figure 1. The components of the receiver for detection, synchronization and demodulation were also developed in MATLAB. Each signal was transmitted 10,000 times and the results were recorded. Additionally, 100 runs of 10,000 noise samples were transmitted into the receiver to determine the number of resulting false alarms.

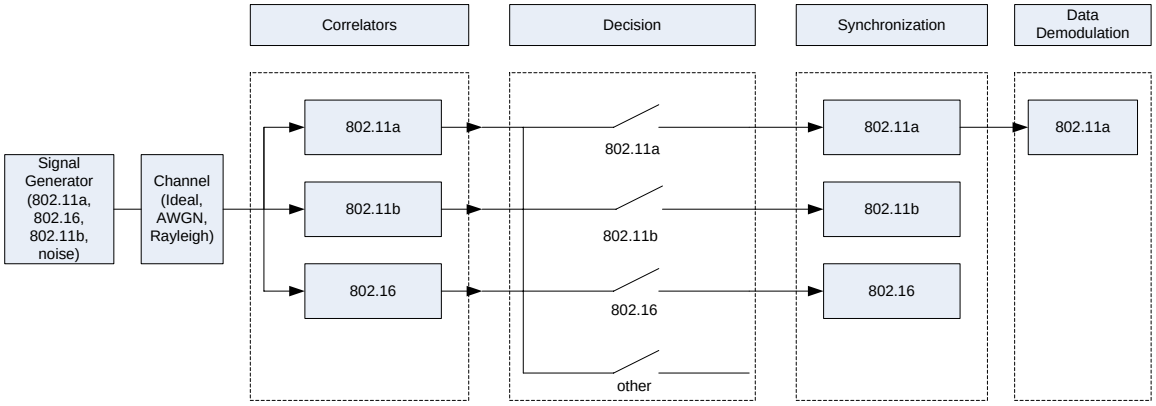


Figure 1. Single Receiver Block Diagram for Multiple Wireless Networking Waveform Detection and Synchronization.

The results of the simulations showed that, over a wide range of signal to noise ratios, the IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 signals could be distinguished from each other. Over the 10,000 runs of each signal and the 100 runs of noise signals, no false alarms were generated by any of the unintended correlators. The implication of this is that a single digital receiver with a wideband air interface capable of sampling each signal can be used to receive multiple wireless networking signals. As more waveforms are to be added to the receiver's capabilities, the first consideration has to be the cross-correlation prosperities of the new signal's preamble with the preambles of the other signals. The only addition to the receiver is another software module, assuming that the new signal's frequency band is within the current operating range of the air interface. The results also showed that the cross-correlation result can be used very effectively to achieve frame synchronization for each of the signals tested.

Another result of the simulations concerned the frame information available when the full IEEE 802.11a signal could not be demodulated due to low signal to noise levels. It was noticed that even at low signal to noise levels, where the probability of parity check failure was high, the bit error rates within the frame header field were low. Furthermore, it was found that the data rate and data length field of the header could be read correctly with significantly high probability of correctness even when the overall signal could not be demodulated.

ACKNOWLEDGMENTS

There are not words to express my gratitude for the support and patience my wife offered me during this process.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

With the popularity of IEEE 802.11a/b/g wireless networks, the prevalence of cellular networks, and the introduction of the IEEE 802.16 standards, a constantly changing selection of wireless networking choices will be available to the mobile user. As the number of wireless networks increases, a radio device capable of connecting to multiple wireless networks will be necessary. The device will need to be able to detect and classify different wireless networking signals and to distinguish the signals from each other. The device must also decide which network to connect to depending upon select criteria, such as data rate or cost. One implementation towards this approach is to include a separate receiver for each communication standard desired. This requires an ever growing footprint as the number of communication options increases. Another solution is a receiver with a radio frequency (RF) front end that operates in the frequency ranges of the desired signals. This wideband air interface would sample the signals for digital processing. Once the RF signal had been sampled, the samples would be digitally filtered to determine if any of the signals of interest are present. The receiver would then decide which network connection is most suitable. For this solution to be practicable, the digital receiver must be able to detect and classify each of the signals of interest with a high probability of detection, while at the same time not falsely classifying one signal as another.

The research interest in this thesis is to investigate the detection of IEEE 802.11a, IEEE 802.16, and IEEE 802.11b wireless networking signals by a single digital receiver. IEEE 802.11a and IEEE 802.11b compliant systems operate near 5 GHz and 2.4 GHz, respectively, in the United States [1] [2]. The operating frequencies of the IEEE 802.16 standards range from 2 GHz to 66 GHz [3]. This thesis assumes that the receiver has a wideband air interface capable of receiving each of these signals and sampling them at baseband. The thesis then explores the ability of signal processing techniques to detect, classify, and demodulate the signals.

The Department of Defense (DoD) interest in these technological developments is two fold. First, as the DoD seeks high data-rate, battlefield communications, the underlying physical layer and medium access (MAC) layer technologies implemented by these standards offer significant opportunity for leveraging commercial technology into military communication systems. Second, as these signals become ever more commonplace, the interest will arise to be able to detect, classify and demodulate these signals for signal intelligence purposes.

B. OBJECTIVE AND APPROACH

The objective of this thesis is to determine the suitability of a single, baseband, digital receiver to detect and classify IEEE 802.11a, IEEE 802.16, and IEEE 802.11b signals. Suitability is defined by the probability of detection of each signal when it is present and the probability of false alarm for the other signals when they are not present. The second objective is to implement this receiver using MATLAB and test its performance in a communication channel defined by multipath propagation interference in addition to additive white Gaussian noise. The third objective is to determine the extent to which IEEE 802.11a signals can be demodulated in multipath environments when full signal demodulation fails.

The approach to these objectives will be to analyze the signal detection problem first. The preambles in the IEEE 802.11a and IEEE 802.16 packet headers will be used as the basis of signal detection and classification. To this end, the preamble structure of each of these standards will be analyzed and their auto- and cross-correlation properties investigated. Following this, an attempt is made to define the probability density functions for the presence and absence of an IEEE 802.11a or IEEE 802.16 signal in an additive white Gaussian noise environment. These results will then be used to analytically determine a detection threshold based upon the Neyman-Pearson test from classical signal detection theory. The corresponding probabilities of detection for different decision thresholds will be analyzed. With these results, a receiver operating characteristic (ROC) curve for each standard will be developed. From the ROC curve, a desired balance between a probability of detection and a probability of false alarm can be

found. Following this, techniques for frame synchronization will be investigated utilizing the signal detector output to best determine the beginning of the packet header in the received samples.

All simulations in this thesis will be conducted with MATLAB. A model will be introduced for the generation and demodulation of IEEE 802.11a signals. A second model that partially implements the generation and demodulation of IEEE 802.16 signals will also be introduced. Another model will be developed to simulate different channel environments. These models will be linked to the developed detection, classification, and synchronization model to generate test results concerning the ability of a common, digital receiver to detect, classify, and synchronize to IEEE 802.11a, IEEE 802.16, and IEEE 802.11b signals. Additionally, these results will be analyzed to determine the ability of the receiver to read packet header information in noisy channel environments.

C. RELATED WORK

The literature concerning the determination of decision thresholds for signal detection of OFDM signals is not extensive. The significant paper is [4], where decision thresholds were devised for a 1024-carrier OFDM signal. No work was found concerning the determination of decision thresholds for IEEE 802.11a or IEEE 802.16 signals. This thesis attempts to follow the approach in [4] to determine appropriate decision thresholds for IEEE 802.11a and IEEE 802.16 signals.

Much has been written in the literature on frequency and frame synchronization for OFDM signals. Much of this work has been concerned with OFDM signals in general, such as [5] [6] [7] [4]. More recent work has focused on specific implementation issues of frequency offset correction and frame synchronization of IEEE 802.11a and IEEE 802.16 signals [8] [9]. This thesis will adapt an algorithm from [8] and apply it to IEEE 802.11a and IEEE 802.16 signals for frame synchronization. This algorithm is compared against the technique described in [4].

D. ORGANIZATION

Chapter II covers the characteristics of radio channels defined by multipath propagation. This chapter also investigates the fundamental principles of OFDM signal generation. The chapter ends with a discussion of how OFDM signals were designed to minimize the deleterious effects of multipath propagation. Chapter III discusses the OFDM signal detection problem and how the IEEE 802.11a and IEEE 802.16 preambles can be used to accomplish it. This chapter also includes an analysis of a frame synchronization technique for OFDM signals. Chapter IV first presents the MATLAB simulation model developed for this thesis. The remaining discussion in the chapter presents the detection and synchronization results when the input to the receiver was an IEEE 802.11a, IEEE 802.16, or IEEE 802.11b signal. Chapter V summarizes the conclusions of the thesis and presents opportunities for future work. The appendix contains the MATLAB code used in the simulation.

II. BACKGROUND

Wireless networks employ orthogonal frequency division multiplexing (OFDM) to achieve high data rate transmissions in multipath channels. In the IEEE 802.11a and IEEE 802.16 standards, OFDM signals transmit data using binary-phase shift keying (BPSK), quadrature phase-shift keying (QPSK), or quadrature amplitude modulation (QAM) with forward error correction coding. This chapter will describe how the ideal channel, the additive white Gaussian noise (AWGN) channel, and the multipath channel affect a radio signal. Following this, OFDM signal design and how the OFDM waveform can help mitigate the effects of multipath channel distortion will be discussed.

A. RADIO CHANNEL

The effects of the channel on a transmitted radio signal are statistical in nature and change over time and can be characterized as a linear system, e.g., a filter. The received signal is a result of the transmitted signal convolved with the channel impulse response

$$x(t) = s(t) * h(t), \tag{2.1}$$

where $x(t)$ is the received signal, $s(t)$ is the transmitted signal, $*$ represents linear convolution, and $h(t)$ is the impulse response of the channel.

The simulations in this thesis include three types of channel: ideal, AWGN, and multipath. The ideal channel causes no change to the transmitted signal. Its impulse response is

$$h(t) = \delta(t), \tag{2.2}$$

hence

$$x(t) = s(t). \tag{2.3}$$

Thermal noise, caused by the motion of electrons in the receiver equipment, is a function of system temperature and is the basis for AWGN. A signal received in an ideal channel with AWGN is modeled as

$$x(t) = s(t) + n(t) \tag{2.4}$$

where $n(t)$ is a Gaussian random process with zero mean.

Multipath channels take into account the attenuation of the signal due to propagation loss and multipath reflections. Two types of signal fading caused by a multipath channel are flat fading and frequency-selective fading. In flat fading and frequency-selective fading, the channel is modeled as time-invariant and the effects of the multipath reflections on the received signal power are considered [10]. In Figure 2, as a radio signal propagates from the transmitter, the electromagnetic wave will reflect, scatter, or diffract off objects. The propagation loss of each path will differ because of the different path lengths from the receiver to the transmitter. Further, the energy absorbed due to the signal's interaction with the objects along a given path will attenuate the signal power. Additionally, since the lengths of the paths to the receiver are different, the received waves will have different phases when they reach the receiver. These different phases, when summed at the receiver, will add constructively or destructively and cause the received signal power to vary. Thus, the total received signal can be modeled as [11]

$$x(t) = \sum_n \alpha_n(t) s[t - \tau_n(t)] + n(t) \tag{2.5}$$

where $\alpha_n(t)$ is the attenuated amplitude of a signal path and $\tau_n(t)$ is the delay of the signal path.

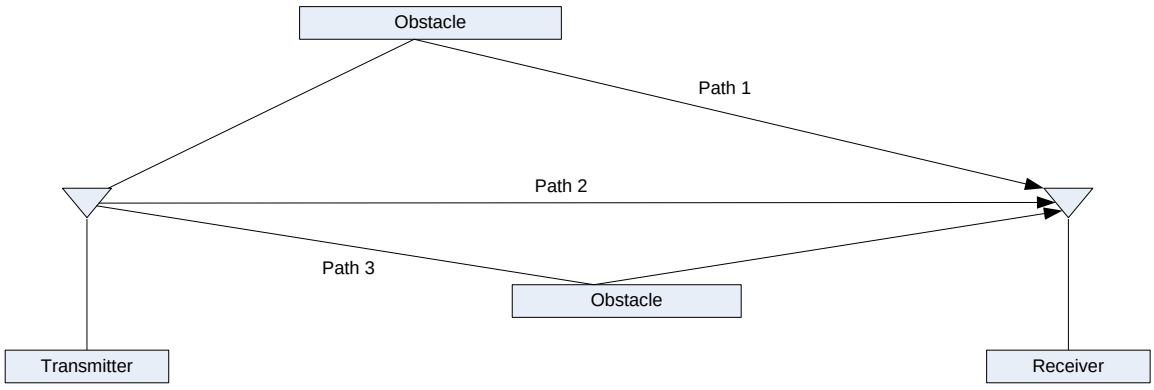


Figure 2. Multipath Channel Environment.

From (2.5), the received signal in a multipath channel is a function of the time delay of the various signal components received over a given interval. Multipath channel delay is typically characterized by the root mean square (rms) delay or delay spread [10]

$$\sigma_\tau = \sqrt{\overline{\tau^2} - (\overline{\tau})^2} \tag{2.6}$$

where

$$\overline{\tau} = \frac{\sum_n a_n{}^2 \tau_n}{\sum_n a_n{}^2} \tag{2.7}$$

and

$$\overline{\tau^2} = \frac{\sum_n a_n{}^2 \tau_n{}^2}{\sum_k a_n{}^2} \tag{2.8}$$

and $a_n{}^2$ is the signal power of the n^{th} path and τ_n is the delay of the n^{th} path .

Channel coherence is a statistical measure of the range of frequencies over which the channel has a certain degree of correlation. If the frequency correlation is required to be above 0.5, the coherence bandwidth is defined as [10]

$$B_c = \frac{1}{5\sigma_\tau}. \tag{2.9}$$

Coherence bandwidth then defines the frequency range over which the channel frequency response will change less than 50% on average. However, if the signal bandwidth is larger than the coherence bandwidth, then the signal will experience frequency-selective fading. In the time domain, frequency-selective fading occurs when

$$T_s < \sigma_\tau, \tag{2.10}$$

where T_s is the symbol time. Under this condition, delayed versions of the signal will cause the received signal to be distorted. The frequency domain representation of the signal will show certain parts of the spectrum having larger gains than others. The effect of this is two fold and is shown in Figure 3. First, the received symbol has less energy because the energy of some of the delayed paths is not captured within the symbol period. Second, this delayed energy from the first symbol is received during the second symbol period and acts as interference. This effect is termed intersymbol interference (ISI) and results in an increased bit error rate (BER). Thus, a multipath channel limits the data rate achievable.

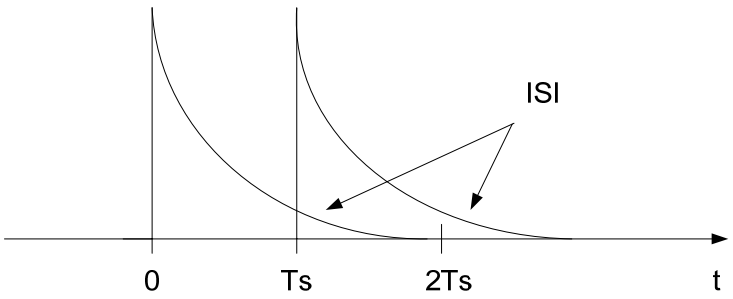


Figure 3. Channel Impulse Response with Intersymbol Interference.

B. OFDM IN A MULTIPATH CHANNEL

For a wireless networking system that requires a high data transmission rate, the multipath channel may limit the effective data rate below what is desired. OFDM signals can overcome this limitation by dividing the spectrum into subchannels. The OFDM transmitter divides its high data rate stream into smaller parallel substreams with symbol periods greater than the channel delay spread. From Table 1, an IEEE 802.11a signal has a bandwidth of 20 MHz and divides the spectrum into 64 subchannels so that each subchannel has a bandwidth of 312.5 kHz. The maximum allowed rms delay spread for a coherent bandwidth of 20 MHz is 10 ns. The maximum allowed rms delay spread for a coherent bandwidth of 312.5 kHz is 640 ns. From Table 1, only 48 of the IEEE 802.11a subchannels are used to carry data. The resulting data rate of each subchannel is then 1/48th of the total system data rate. This means that the effective symbol time per subchannel is 48 times longer than the effective symbol time without frequency division multiplexing. Thus, an OFDM signal can tolerate a greater rms delay spread before being affected by ISI and achieve higher data rate transmissions.

Parameter	Value
N_{SD} : Number of data subcarriers	48
N_{SP} : Number of pilot subcarriers	4
N_{ST} : Number of subcarriers, total	52
Δ_F : Subcarrier frequency spacing	0.3125 MHZ(= 20 MHz/64)
T_{FFT} : IFFT/FFT period	3.2 μs ($1/\Delta_F$)
$T_{PREAMBLE}$: PLCP preamble duration	16 μs
T_{SIGNAL} : Duration of the SIGNAL BPSK-OFDM symbol	4.0 μs ($T_{GI} + T_{FFT}$)
T_{GI} : GI duration	0.8 μs (T_{FFT}/4)
T_{GI2} : Training symbol GI duration	1.6 μs (T_{FFT}/2)
T_{SYM} : Symbol interval	4.0 μs ($T_{GI} + T_{FFT}$)
T_{SHORT} : Short training sequence duration	8.0 μs ($10 \times T_{FFT}/4$)
T_{LONG} : Long training sequence duration	8.0 μs ($T_{GI2} + 2 \times T_{FFT}$)

Table 1. IEEE 802.11a Timing-related parameters (from Ref [1]).

A concern with transmitting the 64 subchannels in parallel is using the bandwidth efficiently. Bandwidth is employed efficiently, if the signals on each subchannel are mutually orthogonal. Orthogonality for two complex signals, $s_1(t)$ and $s_2(t)$, is defined as [11]

$$\int_0^{T_s} s_1(t) s_2^*(t) dt = 0 \qquad (2.11)$$

where * denotes complex conjugation. Orthogonal signals are spectrally efficient because they can overlap in the frequency domain without causing interchannel interference (ICI). Figure 4 shows an ideal, power spectral density plot of three orthogonal subchannels. When each subchannel is at its peak power in the frequency domain, the received power due to the other signals is zero. By taking samples at the peak of the subchannel, the signal energy from only one of the frequencies is captured, and the message on each subchannel will be recovered.

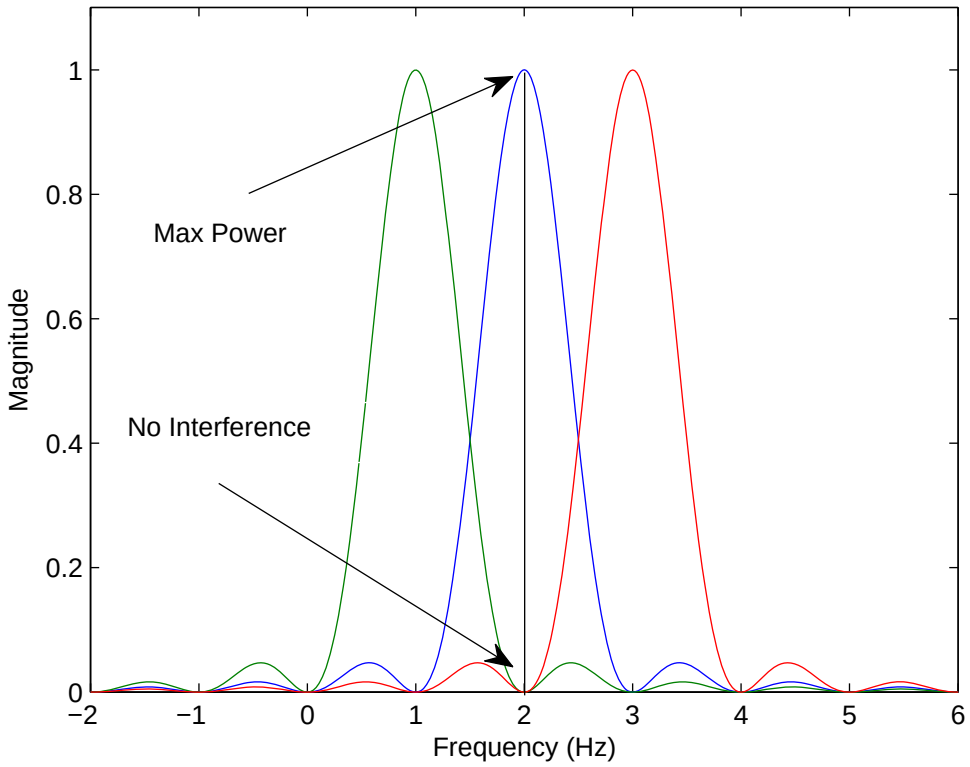


Figure 4. Ideal, Normalized Power Spectral Density Plot of Three Orthogonal Signals.

In the IEEE 802.11 and IEEE 802.16 standards, subchannels are orthogonalized through the use of the inverse discrete Fourier transform (IDFT). The IDFT is defined as [13]

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k] e^{j(2\pi/N)kn} \tag{2.12}$$

where N is the length of the IDFT window, $X[k]$ is the frequency domain input, and $x[n]$ is the time domain sample. The IDFT basis functions $e^{j(2\pi/N)kn}$ are mutually orthogonal [13]:

$$\sum_{n=0}^{N-1} e^{j2\pi kn/N} (e^{j2\pi ln/N})^* = \begin{cases} \dfrac{1-e^{-j2\pi(l-k)}}{1-e^{-j2\pi(l-k)/N}} = 0 & \text{if } l \neq k \\ N & \text{if } l = k \end{cases} \tag{2.13}$$

In IEEE 802.11a, the input sequence to the IDFT is the discrete samples of BPSK, QPSK, or QAM data symbols. From Table 1, the IDFT modulates these samples onto 48 orthogonal subcarriers, where each subcarrier has a bandwidth of 312.5 kHz. The output of the IDFT is 64 time domain samples. When these 64 time domain samples are transmitted in a 3.2-μs period, their frequency domain representation is 64 orthogonal subcarriers (48 data subcarriers, 4 pilot subcarriers, and 12 null subcarriers) occupying a bandwidth of 20 MHz.

Since the IDFT is a periodic function, the output sequence repeats in time [13]. If the output sequence is extended in time cyclically, the frequency domain representation does not change. Samples from the end of the output sequence can be added to the beginning of the output without changing the frequency domain representation. The signal is extended in time but not in frequency. In IEEE 802.11a, sixteen samples are copied from the end of each OFDM symbol and prepended to the beginning of the symbol. These samples extend each OFDM symbol in time by $0.8\ \mu s$. By adding these samples to the beginning of the symbol, a guard interval or cyclic prefix (CP) is created.

These sixteen time domain samples protect the signal against ISI by lengthening the effective symbol time on each subchannel. If the energy from one symbol extends into the next due to multipath reflections, these sixteen samples will be corrupted. However, since these samples are redundant, the receiver can discard them without loss of information. Thus, multipath reflections that arrive less than $0.8\ \mu s$ into the next symbol will only corrupt the cyclic prefix and no information will be lost.

This chapter discussed the channel models that will be used in the thesis and how OFDM signals are designed to allow for high data rate transmissions in multipath channels. The next chapter will discuss signal detection and frame synchronization techniques for OFDM signals.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SIGNAL DETECTION, CLASSIFICATION & SYNCHRONIZATION

Every radio receiver must be able to distinguish signals from noise. Once a receiver has determined that a signal is present, it must determine if the signal is intended for the receiver. If the signal is not classified as one intended for the receiver, it will be ignored. If the signal is classified as one intended for the receiver, the receiver will attempt to demodulate it. As shown in Figure 5, the receiver makes the detection determination by comparing the value of the output of a correlator, a random variable, against a pre-determined decision threshold. If the detector output is higher than the threshold, detection occurs. This chapter outlines how the decision threshold is determined and how the detector output is computed.

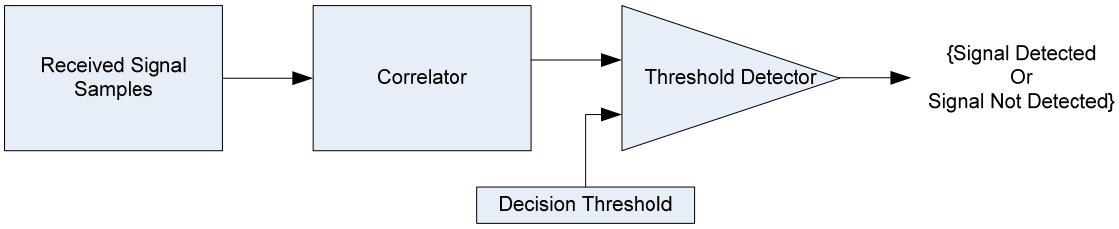


Figure 5. Correlator-based Signal Detector.

A. DETECTION AND CLASSIFICATION

In classic binary detection theory, the receiver must choose between two hypotheses, H_1 and H_0. The first hypothesizes that symbol 1 has been sent. The second hypothesizes that the symbol 0 has been sent. A probability density function (PDF) can be used to model both hypotheses. This problem can also be presented as detecting the presence or absence of a desired signal, which is typical of a radar target detection scenario. In this case, the first hypothesis assumes the signal is present and second hypothesis assumes that the signal is absent and only noise is present. Since the channel introduces distortion into the received signal, the first hypothesis is characterized by the statistical nature of the channel and the power of the signal. The second hypothesis is characterized by the statistical nature of the noise.

When the decision between the two hypotheses is based upon the ratio of the PDFs being larger or smaller than a given threshold, the decision test is called the likelihood ratio test [14],

$$\frac{f_1(x)}{f_0(x)} \underset{H_0}{\overset{H_1}{\gtrless}} \frac{P_0(C_{10}-C_{00})}{P_1(C_{01}-C_{11})} \tag{3.1}$$

where $f_1(x)$ is the PDF of the hypothesis that assumes the signal is present, $f_0(x)$ is the PDF of the hypothesis that assumes only noise is received, P_1 and P_0 are the a priori probabilities of the signal being present or not, and C_{ij} for $i = 0,1$ and $j = 0,1$, are the costs associated with each course of action.

When the a priori probabilities are not known or it is not practical to assign costs, the Neyman-Pearson test is used. The Neyman-Pearson test sets a constraint on the probability of false alarm, such that [15]

$$P_f = \int_{x_T}^{\infty} f_0(x)dx = \alpha' \tag{3.2}$$

where α' is the constraint (a constant) and x_T is the threshold. A constraint is chosen and (3.2) is solved for x_T. The probability of false alarm is the likelihood that the decision statistic based upon noise input will be above the threshold. Once a threshold has been determined, then the probability of detection, P_d, and the probability of miss, P_m, can be found using the PDF of the signal being present [15]:

$$P_d = \int_{x_T}^{\infty} f_1(x)dx \tag{3.3}$$

and

$$P_m = \int_{-\infty}^{x_T} f_1(x)dx. \tag{3.4}$$

The probability of detection is then the likelihood that the decision statistic generated when the signal is present is above the threshold. Conversely, the probability of miss is the likelihood that the decision statistic generated when the signal is present is below the threshold. Figure 6 depicts the PDFs of a two-hypothesis scenario in an

AWGN channel. As the threshold is raised, P_f and P_d decrease, and P_m increases. The goal of signal detection is to make P_f as small as possible while maintaining an acceptable P_d.

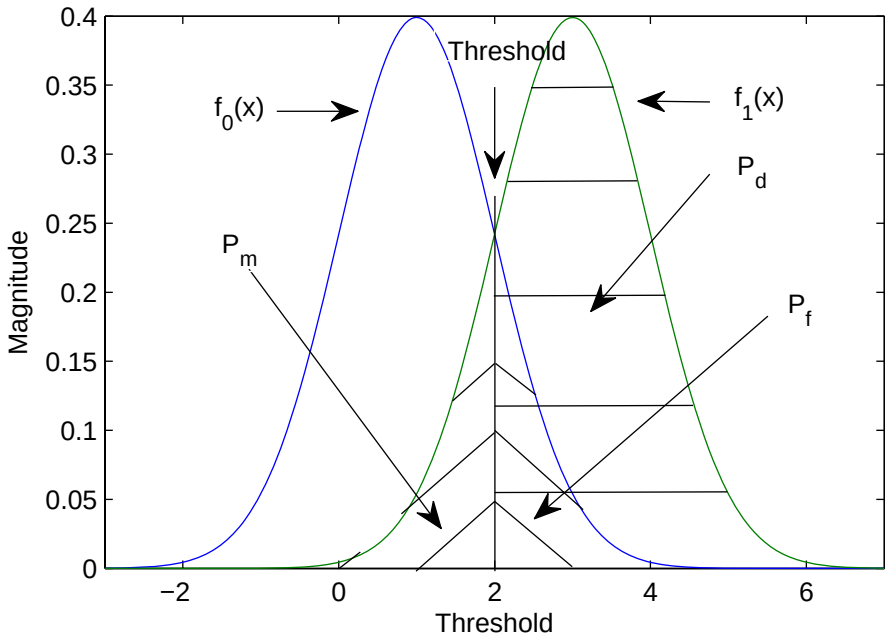


Figure 6. Conditional Probability Density Curves Showing the Areas Defining P_d, P_f, and P_m.

1. IEEE 802.11a and IEEE 802.16 Preambles

High data rate, packet-based wireless local area networks (WLANs) require fast detection and classification times because of the bursty nature of packet-based communications. Without a continuous signal to maintain synchronization, WLANs must synchronize quickly to avoid suffering a lower data rate due to the overhead needed for synchronization. Detection of IEEE 802.11a and IEEE 802.16 signals is performed by searching for the preamble in a received signal by correlating the received signal against one symbol of the short or first preamble, respectively.

In discrete time, the cross-correlation between two signals, $x[n]$ and $y[n]$, is defined as

$$C[k] = \sum_{n=0}^{L-1} x^*[n+k]\, y[n+k-L+1]\,. \tag{3.5}$$

where L is the size of the sliding window of the correlator and * denotes complex conjugation. The power in one of the signals

$$P[k] = \sum_{n=o}^{L-1} \left| x[n] \right|^2 \tag{3.6}$$

can be used to normalize the cross-correlation such that

$$D[k] = \frac{C[k]}{P[k]}\,. \tag{3.7}$$

The cross-correlation result is normalized so that a single decision threshold can be used regardless of the received signal power.

For the preambles to be useful in achieving signal detection in a joint detection receiver, they must have low normalized cross-correlation values. The following sections will detail the structure of the IEEE 802.11a and IEEE 802.16 preambles and their correlation properties.

a. IEEE 802.11a Preambles

Both the IEEE 802.11a and IEEE 802.16 standards begin a frame transmission with a preamble. The preamble is intended to be used for signal detection, classification, and time and frequency synchronization. In both standards, the preamble, or training sequence as it is also called, is broken into two parts.

In IEEE 802.11a, the two parts are called the short preamble and the long preamble and are depicted in Figure 7. The short preamble consists of 10 short symbols and the long preamble consists of two long symbols. The 10 short symbols of the short preamble are comprised of 12 subcarriers of an OFDM symbol [1]. The transmission time of each of the short symbols is $0.8\ \mu s$. After sampling at 20 MHz, there are 16 time domain samples per short symbol. The short preamble then consists of a total of 160 time domain samples lasting $8.0\ \mu s$.

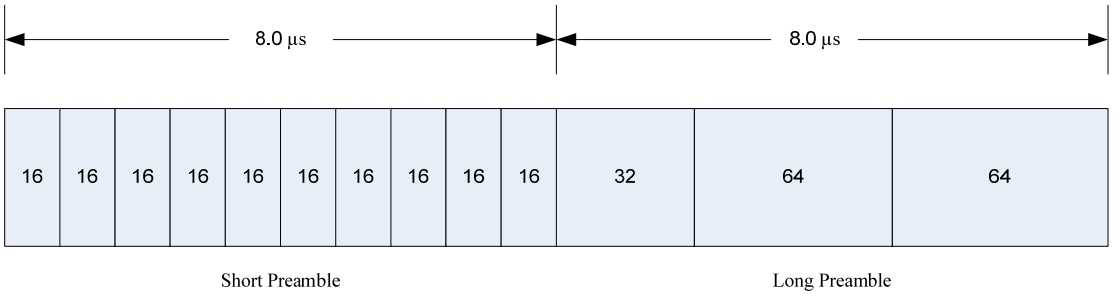


Figure 7. IEEE 802.11a Short and Long Preambles (From Ref [1]).

Table 2 gives the standard time domain sample values of one of these short symbols. Both the real and the imaginary samples sum to zero and each has seven positive samples, eight negative samples and one zero term. Thus, when these samples are multiplied by a random input, the resulting magnitude will be at least an order of magnitude smaller than the original input. Further, since half the terms are positive and the other half negative, when the results of the multiplications are added together, the sum will again be smaller. Thus, when the input samples directly match these values, the magnitude of the correlated output is expected to be significantly higher than otherwise.

IEEE 802.11a Short Preamble Time Domain Samples	
Real Time Domain Samples	Imaginary Time Domain Samples
0.046	0.046
-0.1324	0.0023
-0.0135	-0.0785
0.1428	-0.0127
0.092	0
0.1428	-0.0127
-0.0135	-0.0785
-0.1324	0.0023
0.046	0.046
0.0023	-0.1324
-0.0785	-0.0135
-0.0127	0.1428
0	0.092
-0.0127	0.1428
-0.0785	-0.0135
0.0023	-0.1324

Table 2. Short Preamble Time Domain Sequence Samples.

The first six symbols of the short preamble are intended to be used for signal detection, automatic gain control, and diversity selection. The last four short symbols are intended for coarse frequency offset estimation and timing synchronization [1].

The long preamble consists of a cyclic prefix and two long symbols. In the time domain, each long symbol consists of 64 complex time domain samples when sampled at 20 MHz. The cyclic prefix is the last 32 samples of the symbol prepended to the first long symbol. The cyclic prefix is transmitted for 1.6 μs and each long symbol is transmitted for 3.2 μs. The total length of the long preamble is then 160 samples with a transmission time of 8.0 μs. The long preamble is intended for channel estimation and fine frequency offset estimation [1]. Figure 8(a) shows one symbol of the short preamble correlated with the full IEEE 802.11a preamble in an ideal channel. After an initial sequence of twenty zeros, the ten symbols of the short preamble each create peaks that are 80% above the cross-correlation between short and long preamble. In Figure 8(b), the cross-correlation of one symbol of the long preamble with the full IEEE 802.11a preamble results in one smaller peak that marks the extended cyclic prefix and then two taller peaks that are 3.8 times greater than the cross-correlation between the long and short preamble symbols.

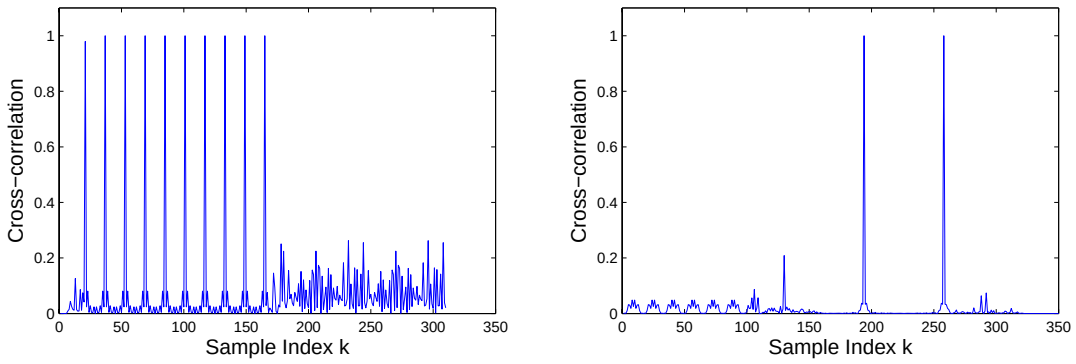


Figure 8. Cross-correlation of the IEEE 802.11a Preamble with (a) Short Preamble Correlator and (b) Long Preamble Correlator.

b. IEEE 802.16 Preambles

In the IEEE 802.16 standard, the preambles are different on the downlink and the uplink. The uplink preamble is a shortened version of the downlink preamble. Here, the downlink preamble as shown in Figure 9 will be described. The first symbol consists of a cyclic prefix followed by four repetitions of a 64 sample sequence. The second symbol is comprised of a cyclic prefix followed by two repetitions of a 128 sample sequence. Both cyclic prefixes are the same length and each is also the same length as the cyclic prefixes within the OFDM data symbols. In IEEE 802.16, the cyclic prefix size is variable, depending upon the channel rms delay spread.

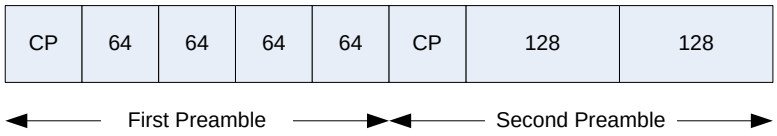
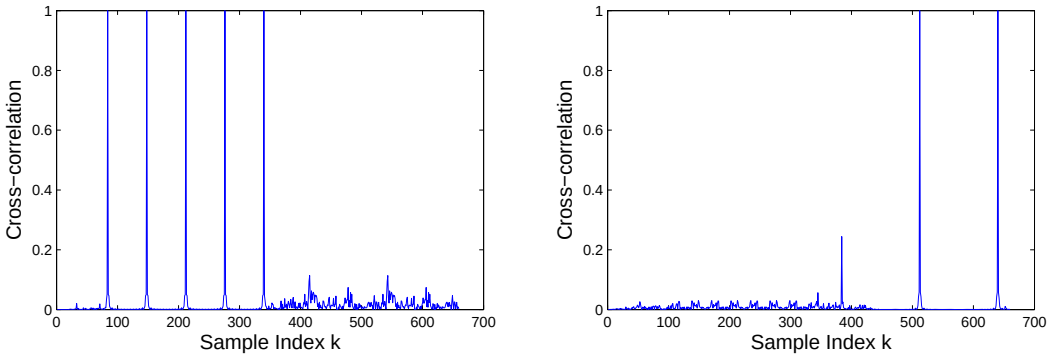


Figure 9. IEEE 802.16 Downlink Preamble (From [12]).

Using a cyclic prefix of 64 time domain samples, Figure 10(a) shows the cross-correlation of the complete downlink preamble with one symbol of the first preamble. There are five peaks because the cyclic prefix matches the length of a symbol of the first preamble. The cross-correlation between one symbol of the first preamble and one symbol of the second preamble is 8.8 times less than the maximum auto-correlation of one symbol of the first preamble. Figure 10(b) shows the cross-correlation of one symbol of the second preamble with the complete downlink preamble. The smaller correlation peak results from the cyclic prefix. The maximum cross-correlation value between one symbol of second preamble and one symbol of the first preamble is more than seventeen times less than the max autocorrelation value of one symbol of the second preamble.

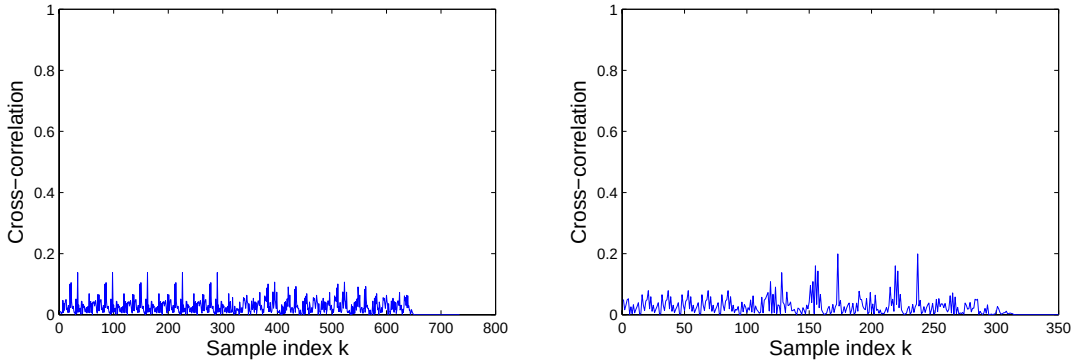


(a) (b)

Figure 10. Cross-correlation of the IEEE 802.16 Preamble with (a) First Preamble Correlator and (b) Second Preamble Correlator.

2. Cross-correlation of IEEE 802.11a and IEEE 802.16 Preambles

Figure 11(a) shows the results of running the IEEE 802.16 preamble through the IEEE 802.11a short preamble correlator in an ideal channel. Figure 11(b) shows the results of running the IEEE 802.11a preamble through the IEEE 802.16 first preamble correlator in an ideal channel. Table 3 lists the maximum cross-correlation values of the IEEE 802.11a and IEEE 802.16 preamble symbols. As can be seen from both Figure 11 and Table 3, in an ideal channel an IEEE 802.16 signal will only trigger a detection in an IEEE 802.11a receiver if the decision threshold is set below 0.1379. Furthermore, in the same channel an IEEE 802.11a signal will only trigger a detection in a IEEE 802.16 receiver if the decision threshold is set below 0.1991.



(a) (b)

Figure 11. Preamble Cross-correlation: (a) IEEE 802.11a Short Preamble Correlator Output with IEEE 802.16 Downlink Preamble Input; (b) IEEE 802.16 First Preamble Correlator Output with IEEE 802.11a Preamble Input.

Correlator and Input	Maximum Value
802.16 preamble through 802.11a short preamble correlator	0.1379
802.16 preamble through 802.11a long preamble correlator	0.0460
802.11a preamble through first 802.16 preamble correlator	0.1991
802.11a preamble through second 802.16 preamble correlator	0.0800

Table 3. Maximum Cross-correlation Values between IEEE 802.11a and IEEE 802.16 Preamble Symbols.

3. Decision Statistics

This section will compare the decision statistics from two correlation methods used for signal detection. The first method is the autocorrelation method and is detailed in [4]. The second method is the cross-correlation method and will be explained below. For each method, the mean and the variance of the decision statistic will be determined when the signal is present and when only noise is present. Following this, the statistical distribution for the noise only case will be discussed in order to apply the Neyman-Pearson test to determine detection thresholds.

In [4], a PDF for the noise-only hypothesis was derived for an OFDM signal using autocorrelation to detect a repeated sequence. This paper, written before the IEEE 802.11a standard was adopted, called for a two symbol training sequence. The first symbol was to contain two identical sequences preceded by a cyclic prefix and was to be used for frame synchronization. The second symbol was to contain two separate pseudonoise sequences on the even and odd frequencies to enable frequency offset calculations. The simulations in [4] used an OFDM signal with 1000 subcarriers oversampled at 1024 samples per symbol. The guard interval was equal to 10% of the data symbol. The normalized decision statistic was defined as [4]:

$$D_A[k] = \frac{\left|R[k]\right|^2}{(P[k])^2} \tag{3.8}$$

where the autocorrelation is defined as

$$R[k] = \sum_{n=0}^{L-1} (x[n+k]^* x[n+k-L+1]) . \tag{3.9}$$

Since the real and imaginary parts of the noise are assumed to be Gaussian, the autocorrelation of the noise is a chi-square random variable defined as

$$\left|R[k]\right|^2 = 2L\sigma_n{}^4 \chi_2{}^2 \tag{3.10}$$

where $\chi_2{}^2$ is a chi-square random variable with a mean of two and a variance of four. The square of the received noise power is Gaussian with a mean of $4L^2\sigma_n{}^4$ and a variance of $16L^4\sigma_n{}^8$ [4]. If each term in (3.8) is divided by the mean of the denominator, the distribution of $D_A[k]$ is

$$D_A[k] \sim \frac{\frac{1}{2L}\chi_2{}^2}{n\left(1, \frac{4}{L}\right)} \tag{3.11}$$

where $\sim$ signifies "is distributed" and $n(a,b)$ signifies a Gaussian random variable with mean a and variance b. Since the mean of the denominator is one and the variance is smaller than the mean, the PDF of the decision statistic for the noise-only case can be approximated as [4]

$$D_A[k] \sim \frac{1}{2L}\chi_2{}^2 . \tag{3.12}$$

The mean and variance of the decision statistic are

$$E\left[D_A[k]\right] = E\left[\frac{1}{2L}\chi_2{}^2\right] = \frac{1}{2L}E\left[\chi_2{}^2\right] = \frac{1}{2L}(2) = \frac{1}{L} \tag{3.13}$$

$$\text{var}[D_A[k]] = \frac{1}{4L^2}E\left[(\chi_2{}^2)^2\right] - \frac{1}{L^2} = \frac{\text{var}(\chi_2{}^2) + E\left[\chi_2{}^2\right]}{4L^2} - \frac{1}{L^2} = \frac{4+4}{4L^2} - \frac{1}{L^2} = \frac{1}{L^2} \tag{3.14}$$

where $E[\ ]$ is the expectation operator.

This approach can be applied in an IEEE 802.11a receiver to determine the mean and variance of the decision statistic when a signal is present and when only noise is present. In the following sections, this method will be compared against the mean and variance of the cross-correlation decision statistic.

a. IEEE 802.11a Receiver Cross-Correlation Decision Statistic when Preamble is not Present

In the cross-correlation approach, a cross-correlation is performed between the received samples and one symbol of the IEEE 802.11a short preamble. When the preamble is not present, the received signal is only noise. The cross-correlation is defined in vector notation as:

$$\mathbf{c} = \mathbf{p}^H\mathbf{x} = \mathbf{p}^H\mathbf{n} \tag{3.15}$$

where $\mathbf{p}^H$ is the complex conjugate transpose of one symbol of the short preamble and $\mathbf{n}$ is a vector of sixteen complex noise samples. The assumption is made that the real and imaginary parts of the noise samples are Gaussian random variables with zero mean and a variance of one. Similar to (3.8), the normalized decision statistic is defined in vector notation as:

$$D_C[k] = \frac{|\mathbf{c}|^2}{\left(|\mathbf{p}|^2\right)^2} \tag{3.16}$$

where $|\mathbf{p}|^2 = 0.2031$. The correlation term in the above equation can be expanded as

$$\mathbf{c} = (\mathbf{p}_I + j\mathbf{p}_Q)^H(\mathbf{n}_I + j\mathbf{n}_Q) . \tag{3.17}$$

The numerator of the decision statistic can then be expanded into

$$\begin{aligned}|\mathbf{c}|^2 = (\mathbf{p}_I{}^T\mathbf{n}_I)^2 + 2(\mathbf{p}_I{}^T\mathbf{n}_I)(\mathbf{p}_Q{}^T\mathbf{n}_Q) + (\mathbf{p}_Q{}^T\mathbf{n}_Q)^2 - \\ 2(\mathbf{p}_I{}^T\mathbf{n}_Q)(\mathbf{p}_Q{}^T\mathbf{n}_I) + (\mathbf{p}_Q{}^T\mathbf{n}_I)^2 + \left(\mathbf{p}_I{}^T\mathbf{n}_Q\right)^2\end{aligned} \tag{3.18}$$

where ^{T} is the transpose of the vector. Taking the expectation of this term yields

$$\mathrm{E}\left\{|\mathbf{c}|^2\right\} = \mathrm{E}\left\{(\mathbf{p}_I{}^T\mathbf{n}_I)^2 + (\mathbf{p}_Q{}^T\mathbf{n}_Q)^2 + (\mathbf{p}_Q{}^T\mathbf{n}_I)^2 + (\mathbf{p}_I{}^T\mathbf{n}_Q)^2\right\} \tag{3.19}$$

where $\mathrm{E}\left\{\mathbf{p}_I{}^T\mathbf{n}_I\mathbf{p}_Q{}^T\mathbf{n}_Q\right\} = 0$ and $\mathrm{E}\left\{\mathbf{p}_I{}^T\mathbf{n}_Q\mathbf{p}_Q{}^T\mathbf{n}_I)\right\} = 0$ because the noise samples are independent, zero-mean random variables and all of the preamble terms are constants. This equation can be further simplified by noting that the expectation of each of the individual terms in (3.16) is equal. Taking one term,

$$\mathrm{E}\left\{(\mathbf{p}_I{}^T\mathbf{n}_I)^2\right\} = \mathrm{E}\left\{\sum_i p_i{}^2 n_i{}^2\right\} = \sum_i p_i{}^2\sigma_n{}^2 \tag{3.20}$$

where $\sigma_n{}^2$ is the variance of the noise samples. Since $\sum_i p_i{}^2 = \sum_i p_q{}^2$, the mean of the cross-correlation term is $4\sum_i p_i{}^2\sigma_n{}^2$. The mean of the decision statistic is then defined as

$$\mathrm{E}\{D_C[k]\} = \frac{4(0.1016)\sigma_n{}^2}{0.0413} = 9.43\sigma_n{}^2. \tag{3.21}$$

When an IEEE 802.11a signal is not present, the decision statistic is based on the power of the received noise. Considering the dynamic range of communication receivers, the assumption is made that the highest acceptable noise power is -20 dBm. As AWGN power is added to the signal, the variance of the decision statistic will increase making it more likely that a random spike will be above the decision threshold and cause a false detection. A low decision statistic, when only noise is present, will help reduce the likelihood of a false detection. In Figure 12, the theoretical values derived above are plotted against simulation results when the input samples are Gaussian with a mean of zero and a standard deviation of σ_n. Ten simulation runs of 10,000 samples were performed with $\sigma_n = 0.01$. The average mean of these runs was taken at every SNR level and plotted against the corresponding SNR. The results are plotted along with the results of the autocorrelation method. The autocorrelation average mean is dependent only on

the correlator window size and is a constant, which for IEEE 802.11a is $\frac{1}{L} = \frac{1}{16} = 0.0625$.

The cross-correlation average mean is dependent upon the input signal power as shown in (3.18). The mean of the cross-correlation decision statistic is over an order of magnitude lower than the autocorrelation method. Thus, the cross-correlation method will be able to tolerate a higher level of additive noise without a higher rate of false detection.

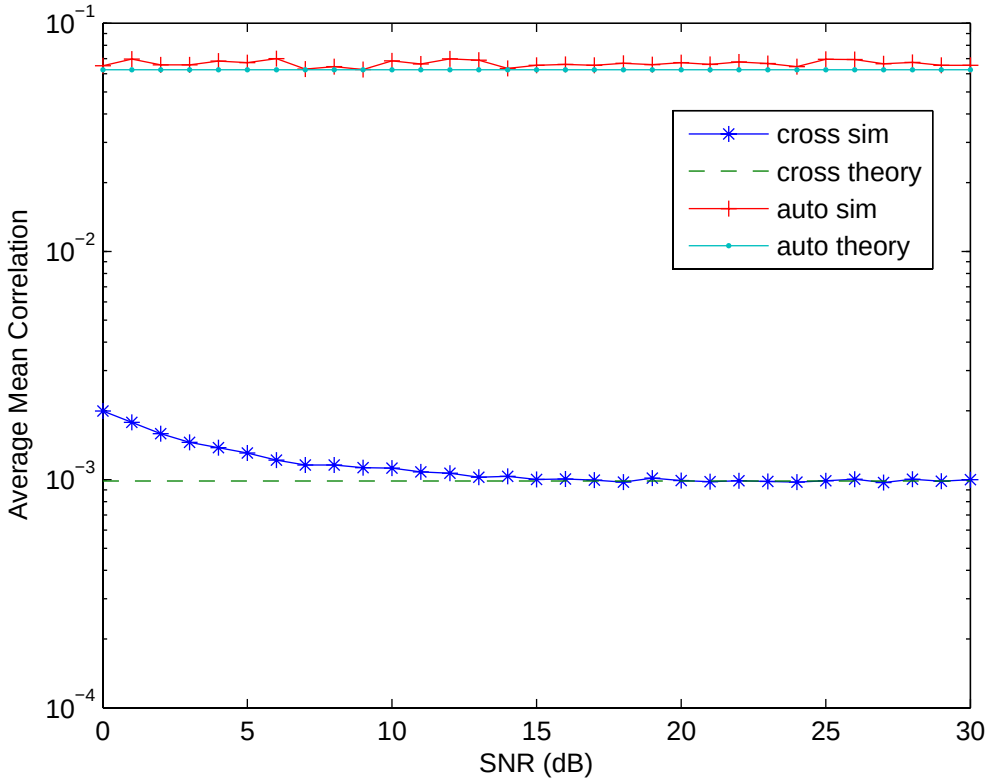


Figure 12. Theoretical and Simulated Average Mean Correlation Values with No Preamble Signal Present for the Autocorrelation and Cross-correlation Methods in an AWGN Channel.

The variance of the cross-correlation term of the decision statistic can be defined as

$$\mathrm{var}(\mathbf{c}) = \mathrm{E}\left\{\left(|\mathbf{c}|^2\right)^2\right\} - \left[\mathrm{E}\left\{|\mathbf{c}|^2\right\}\right]^2. \tag{3.22}$$

From the cross-correlation mean calculations above, it can be seen that the variance of the decision statistic will contain fourth order and second order terms and that the first

order terms will go to zero. Thus the variance is expected to be significantly smaller than the mean. A closed form solution for the cross-correlation variance for IEEE 802.11a could not be derived; the results of [4] are used as theoretical values. For Figure 13, the variances of the two decision statistics were taken 10,000 times at various SNR levels. These values were then averaged and plotted against the corresponding SNR. The simulated results for both methods show the variance to be the square of the mean as predicted in (3.10) for the autocorrelation method. The empirical results suggest that the variance of the cross-correlation can also be characterized as the mean of cross-correlation result squared. The variance of the cross-correlation method is two orders of magnitude lower than the autocorrelation method variance. The impact of a lower variance is a more constant decision statistic, and, thus, one less likely to have additive noise increase its value over the decision threshold and cause a false detection.

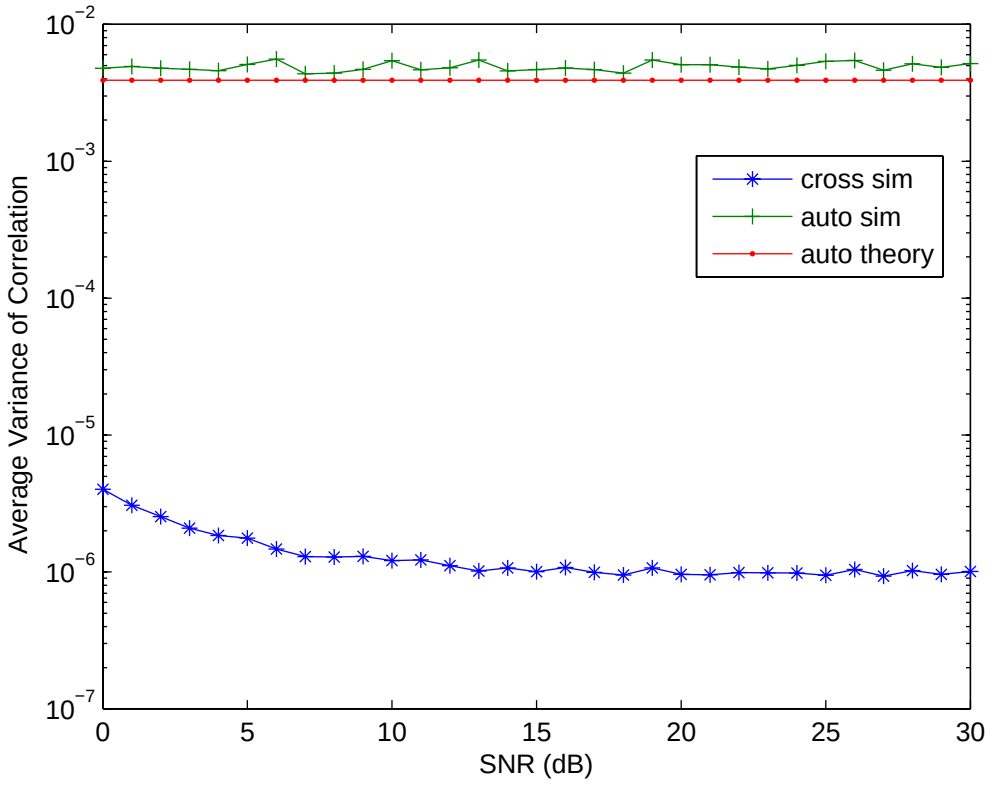


Figure 13. Average Variance of the Autocorrelation and Cross-correlation Methods with No Preamble Signal Present in an AWGN Channel.

b. IEEE 802.11a Receiver Cross-Correlation Decision Statistic when Preamble Signal is Present

When the IEEE 802.11a preamble is present, the decision statistic is still defined using (3.12). The cross-correlation term is defined as

$$\mathbf{c} = \mathbf{p}^H \mathbf{x} = (\mathbf{p}_I + j\mathbf{p}_Q)^H (\mathbf{p}_I + j\mathbf{p}_Q + \mathbf{n}_I + j\mathbf{n}_Q) \tag{3.23}$$

where the received signal is comprised of two vectors: the preamble, $\mathbf{p}$, and AWGN samples, $\mathbf{n}$. Equation (3.20) can be simplified to

$$\mathbf{c} = 0.2032 + \mathbf{p}_I{}^T \mathbf{n}_I + \mathbf{p}_Q{}^T \mathbf{n}_Q - j\mathbf{p}_Q{}^T \mathbf{n}_I + j\mathbf{p}_I{}^T \mathbf{n}_Q . \tag{3.24}$$

The expectation of the magnitude squared is

$$E\left\{|\mathbf{c}|^2\right\} = E\left\{0.2032^2 + (\mathbf{p}_I{}^T \mathbf{n}_I)^2 + (\mathbf{p}_Q{}^T \mathbf{n}_Q)^2 + (\mathbf{p}_Q{}^T \mathbf{n}_I)^2 + (\mathbf{p}_I{}^T \mathbf{n}_Q)^2\right\} \tag{3.25}$$

where all the cross terms are zero as the noise samples are assumed to be uncorrelated. Notice that the last four terms are the same four terms in (3.16). The expected value of the decision statistic is then

$$\mathrm{E}\left\{D_C[k]\right\} = \frac{0.2032^2 + 4(0.1016)\sigma_n{}^2}{0.2032^2} . \tag{3.26}$$

Thus, when the preamble is present, the decision statistic will have a value that varies with the amount of noise power.

In the autocorrelation method, the mean of the decision statistic is defined as [4]

$$\mathrm{E}\left\{D_A[k]\right\} = \mu_{D_A} = \frac{\sigma_s{}^4}{(\sigma_s{}^2 + \sigma_n{}^2)^2} \tag{3.27}$$

where σ_s is the standard deviation of the signal. The mean of this decision statistic decreases with SNR. As the uncorrelated noise power is increased, the autocorrelation between them goes down as shown in Figure 14. The resulting lower mean decision statistic value will cause an increase in the probability of missed detection.

Figure 14 also shows that in the cross-correlation method, the mean of the decision statistic stays centered around one. As the noise power increases, so does the average mean. However, the mean stays closer to one compared to the autocorrelation

method. Thus, a lower probability of missed detections will result. This method has a higher mean when the signal is present because uncorrelated noise is only being added to one term in the correlation rather than both terms as in the autocorrelation method. The mean of the correlation output then remains higher even at lower SNR levels. This results in a lower probability of missed detections.

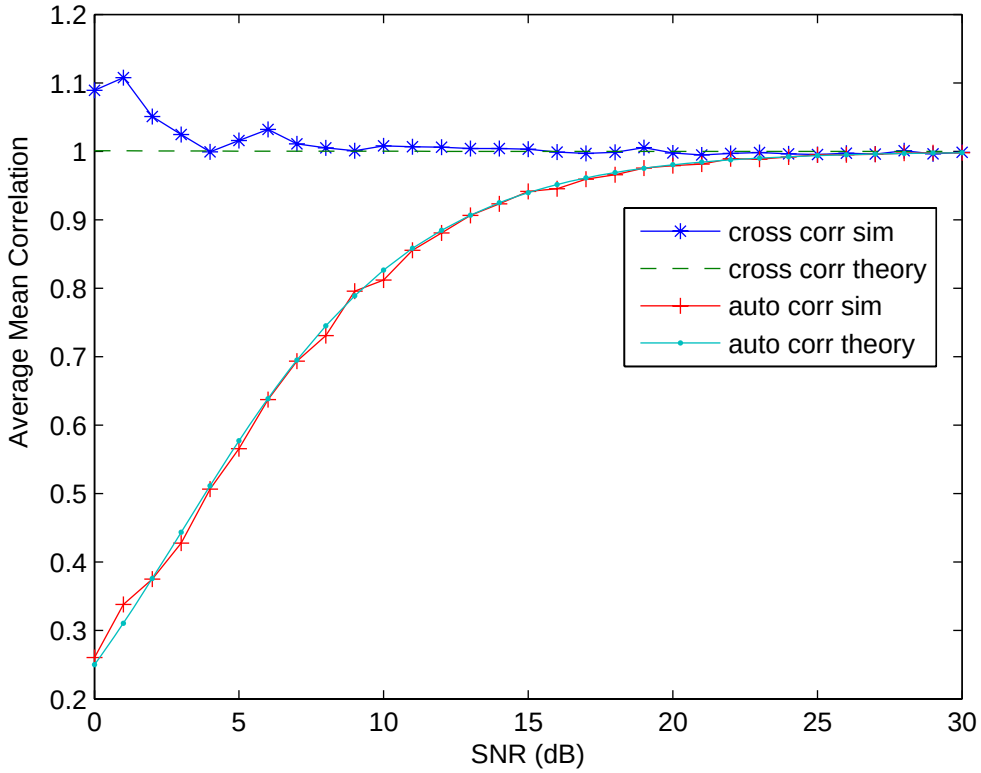


Figure 14. Average Mean of the Autocorrelation and Cross-correlation Methods When an IEEE 802.11a Preamble is Present in an AWGN Channel.

In Figure 15, the cross-correlation output is shown at a SNR of 0 dB. The ten spikes in the figure are the first indices of the ten symbols of the short preamble. Even at 0 dB, the correlation with the preceding 300 noise samples is on an order of magnitude lower than the correlation peaks. Notice that the points between the spikes also remain almost an order of magnitude lower than the correlation peaks. A detector can then count the peaks and the spacing between them to detect and classify the signal.

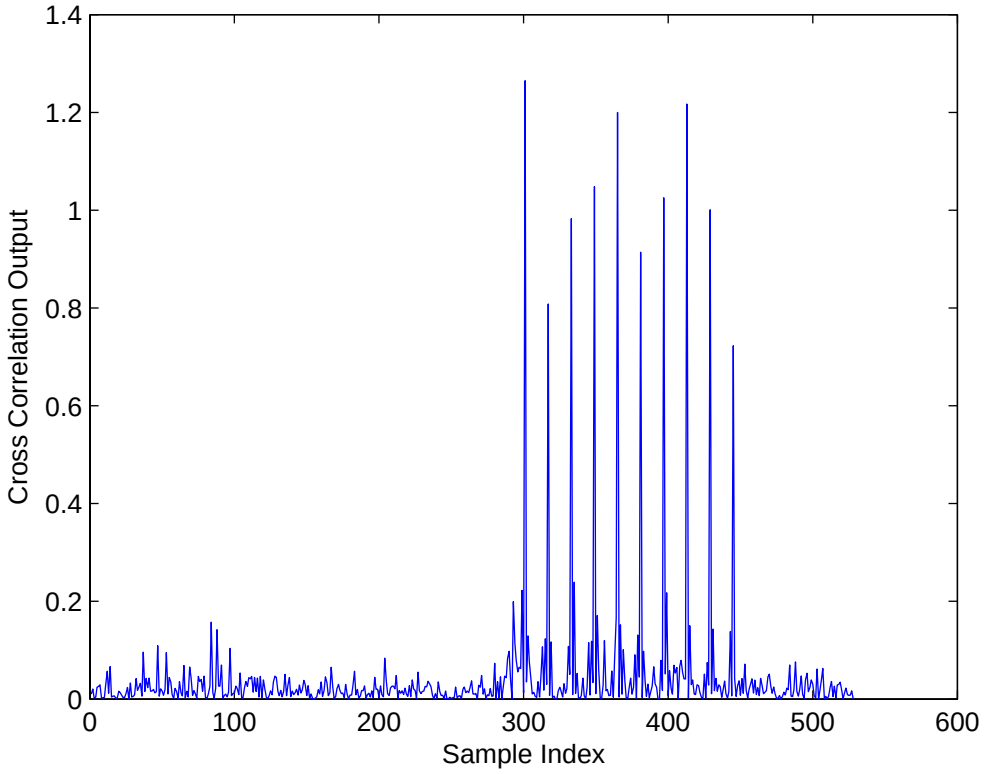


Figure 15. Cross-Correlation with IEEE 802.11a Short Preamble at 0 dB SNR.

In Figure 16, the variance of both methods can be seen to be higher at lower SNR levels. The theoretical autocorrelation variance is defined as [4]

$$\mathrm{var}\left[D_A[k]\right] = \frac{2\sigma_s{}^2[(1+\mu_{D_A})\sigma_s{}^2\sigma_n{}^2 + (1+2\mu_{D_A})\sigma_n{}^4}{L(\sigma_s{}^2+\sigma_n{}^2)^4} \tag{3.28}$$

A closed form solution for the variance of the cross-correlation method for IEEE 802.11a could not be derived. From Figure 16, the variance of both methods decreases as the SNR increases, although the cross-corrlelation variance is higher at lower SNR levels. The difference between the methods is that the mean of the cross-correlation method is around one at the correct timing instants, whereas the variance of the autocorrelation method has an ever decreasing mean as the SNR decreases. Thus, even with a lower variance at lower SNR levels, the autocorrelation method is still expected to underperform the cross-correlation method in poor SNR environments.

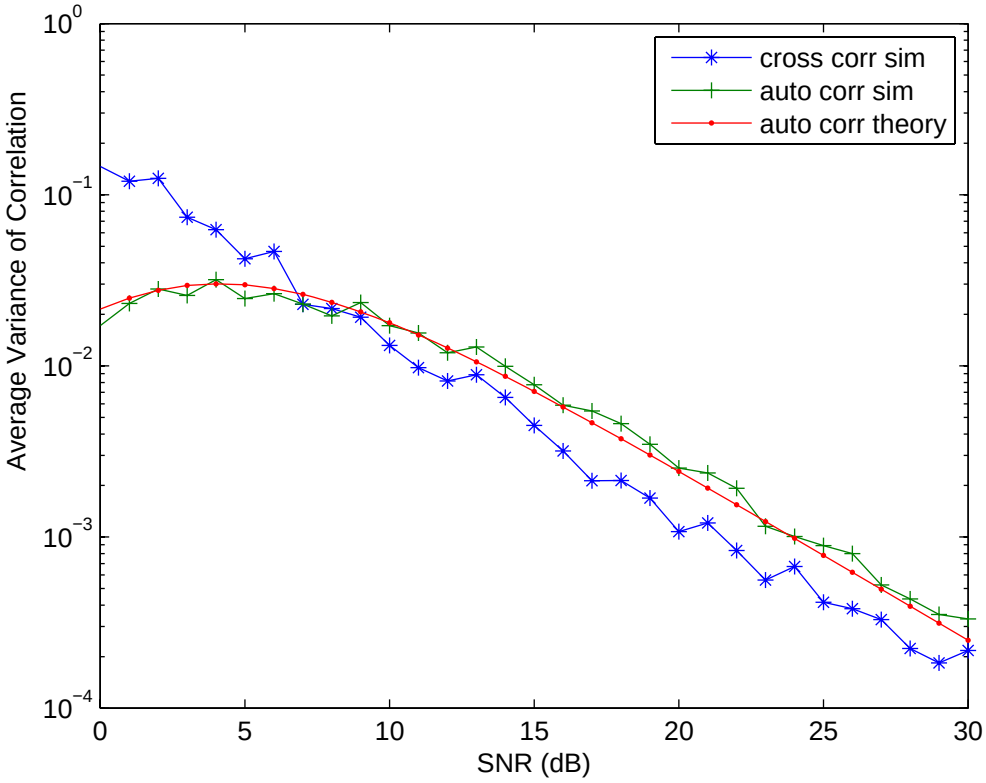


Figure 16. Average Variance of Auto and Cross-correlation Methods When the IEEE 802.11a Preamble is Present in an AWGN Channel.

4. Decision Thresholds

In (3.14), the real and imaginary parts of the noise in $\mathbf{c}$ are zero-mean Gaussian random variables, and $|\mathbf{c}|^2$ is a chi-square random variable because the sum of the square of two zero-mean Gaussian random variables is a chi-square random variable. The mean of this chi square random variable is two and the variance is four [4]. Since $|\mathbf{p}|^2$ is a constant in (3.13), the decision statistic of the cross-correlation method will have the random variable characteristics of $|\mathbf{c}|^2$. Thus, the decision statistic for the hypothesis of the noise only case has a chi-square distribution.

The Neyman-Pearson test can now be applied to determine a decision threshold based upon the PDF of the hypothesis of the noise-only case. To solve the Neyman-

Pearson test, a constraint must be chosen. The constraint details the level of false alarm that is acceptable to the system. In wireless LANs, a high false alarm rate wastes system resources and increases the possibility that an actual signal is missed while processing a false alarm. For mobile nodes on battery power, the energy spent in processing false alarms may be of significance. At the same time, however, a higher false alarm rate necessarily means a higher probability of detection and a lower probability of a missed signal. A surveillance application may be willing to bear the extra processing of false alarms for an increase in detection probability.

In Figure 17, (3.2) is solved over a range of constraints where the distribution used is the chi square distribution derived in [4] with a mean defined as $\frac{1}{L}$ and a standard deviation defined as $\frac{1}{L}$. Correlation windows of 16, 64, and 128 are used. For both, as the threshold increases, the probability of false alarm decreases. As to be expected, the larger correlation window offers better performance.

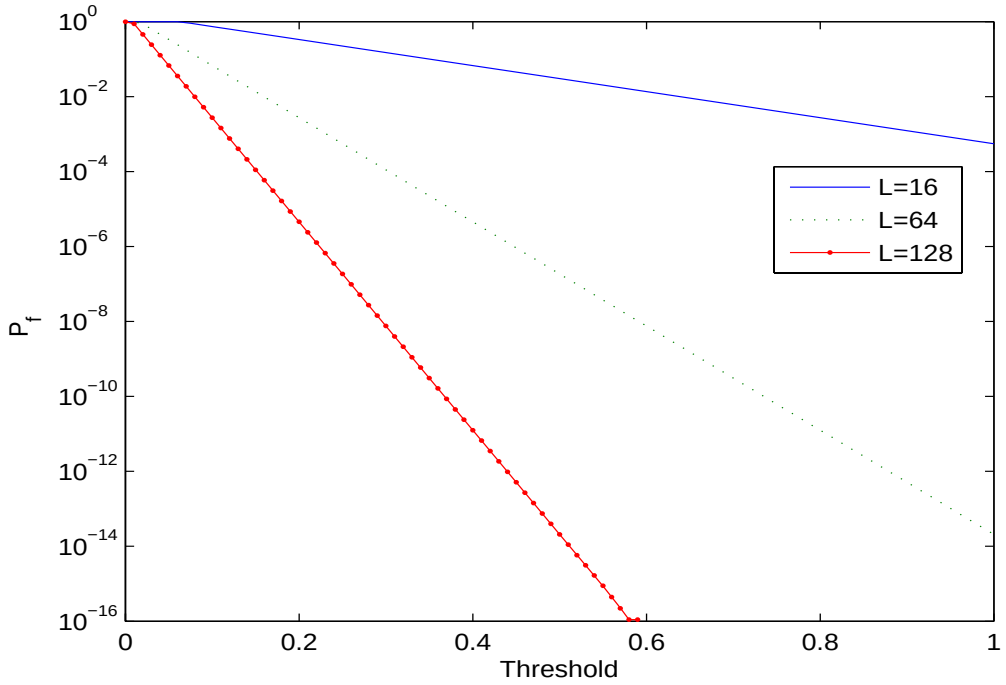


Figure 17. Theoretical Probability of False Alarm as a Function of Decision Threshold for Correlation Window Lengths of 16, 64, and 128.

The thresholds defined from the Neyman-Pearson test can be used to determine the probability of the detection and the probability of miss. The probability of detection is determined using (3.3) where the distribution is Gausssian with mean and variance defined in (3.24) and (3.25), respectively [4]. In Figure 18, the probability of detection is plotted over increasing SNRs for two correlators: one with a correlation window of length sixteen and another with a correlation window of length 64. For the lower SNRs, a very low threshold is needed to maintain a high level of detection. A P_d of 90% at a SNR equal to 1 dB requires a threshold of 0.08. The same P_d at a SNR equal to 30 dB requires a threshold of 0.84. Referring back to Figure 17, the lower threshold for the lower SNR results in a high probability of false alarm. Alternately, at high SNR levels, a higher threshold can be chosen without sacrificing P_d or increasing P_f. Thus, there is always a trade-off between P_d and P_f, but the trade-off is not as significant at higher SNR values.

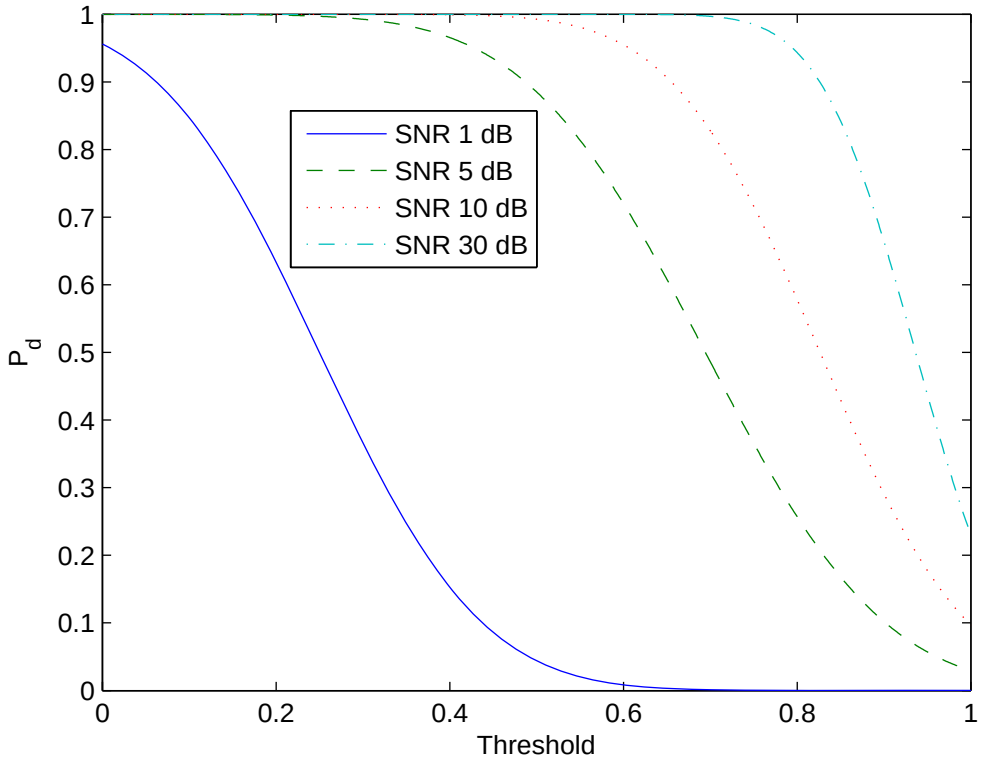


Figure 18. Probability of Detection as a Function of Decision Threshold for Correlation Window Length 16 in an AWGN Channel.

The probability of false alarm and probability of detection for a received signal can be synthesized into one plot called a receiver operator characteristic (ROC) curve. Figure 19 shows the theoretical ROC curves for increasing SNR levels. In this figure, the lower probability of detections can be related to higher probabilities of false alarm. As the SNR increases, the P_{fa} associated with a P_d decreases. As an example, when the SNR = 1 dB, a probability of detection of 0.50 for an IEEE 802.11a signal detector results in a false alarm rate of 0.2230 at a threshold of 0.26. In the ROC curve for an IEEE 802.16 signal detector in Figure 20, the longer correlation window acts in a similar manner to a higher SNR by pushing the curves towards smaller probabilities of false alarm for the same P_d as an IEEE 802.11a receiver. Thus, the ROC curve can be used to find a balance between P_d and P_{fa} in choosing a decision threshold.

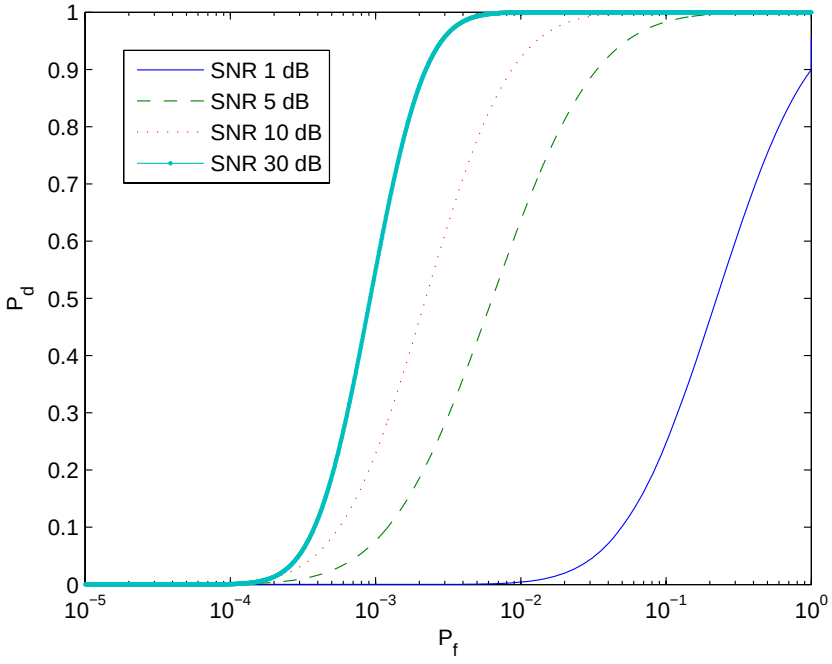


Figure 19. Theoretical Receiver Operating Characteristic Curve for an IEEE 802.11a Receiver using Autocorrelation for Signal Detection in an AWGN Channel.

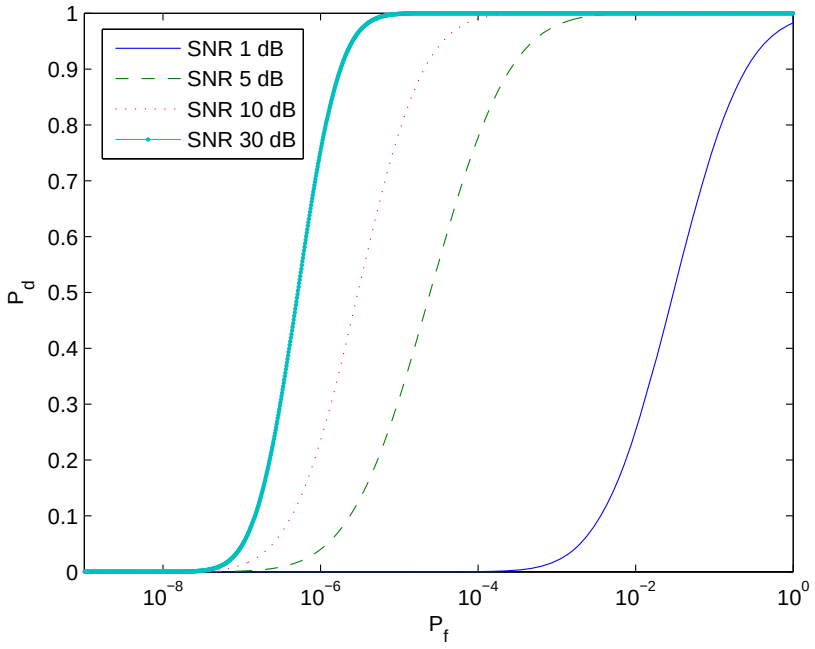


Figure 20. Theoretical Receiver Operating Characteristic Curve for an IEEE 802.16 Receiver using Autocorrelation for Signal Detection in an AWGN Channel.

B. FRAME SYNCHRONIZATION

In both IEEE 802.11a and IEEE 802.16, binary phase-shift keying (BPSK), quadrature phase-shift keying (QPSK), or quadrature amplitude modulation (QAM) is used to transmit data. In each of these modulation schemes, symbols are encoded onto sinusoidal waveforms over defined periods of time. For each of these waveforms to be demodulated, the receiver must find the beginning of the symbol period. If the receiver starts the integration process at the start of the symbol duration, all of the symbol's energy will be captured and the signal will be correctly demodulated. Otherwise, the integration period will include energy from two symbol periods and ISI will result.

In packet-based communications, symbol synchronization depends upon correct frame synchronization. In frame synchronization, the receiver must decide which received time domain sample corresponds to the beginning of the first OFDM symbol following the preamble. An OFDM signal frame is shown in Figure 21. The frame starts with a preamble and is followed by a series of OFDM symbols.

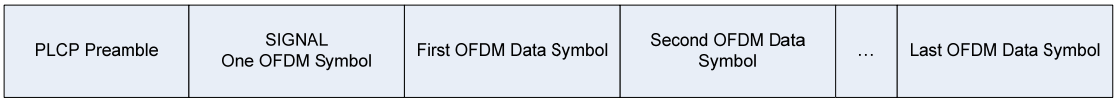


Figure 21. OFDM Signal Frame Format.

Once an OFDM receiver has chosen a time domain sample as the first time domain sample of the first OFDM symbol, the next 80 time domain samples are taken to comprise the OFDM symbol. The first sixteen of these samples are assumed to be the cyclic prefix and are discarded by the receiver. If the time sample chosen is not the first time sample of the cyclic prefix of the first OFDM symbol, then the following 80 samples taken will include data from two OFDM symbols as shown in Figure 22. In this case, the 80 samples will include samples from the following symbol's cyclic prefix. When the FFT is taken of these time domain samples, data from two symbols will have been mixed and ISI will result. Furthermore, since the start of all subsequent OFDM symbols is based upon the timing of the first, the FFT windows of all the following symbols will contain data values from two symbols. Thus, incorrect frame synchronization will cause ISI for all of the OFDM symbols within that frame.

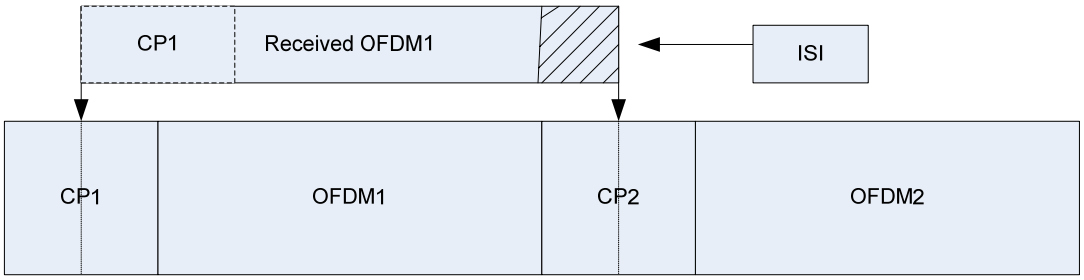


Figure 22. OFDM Symbols without Proper Frame Synchronization.

In [4], frame synchronization is accomplished using the normalized autocorrelation decision statistic. When the signal is detected, the autocorrelation output is a plateau with a length equal to the guard interval as shown in Figure 23. The symbol starting point is found by finding the maximum value within this plateau and then averaging the indices where the autocorrelation value drops below 90% of the maximum [4]. Since the cyclic prefix in IEEE 802.11a signals of the OFDM symbol is not the same

size as the cyclic prefix of the preambles, this technique is not effective for IEEE 802.11a. Since the short preamble is ten symbols of length sixteen, the resulting plateau would be 128 samples long. This is shown in Figure 23 at a SNR of 20 dB with a decision threshold of 0.9.

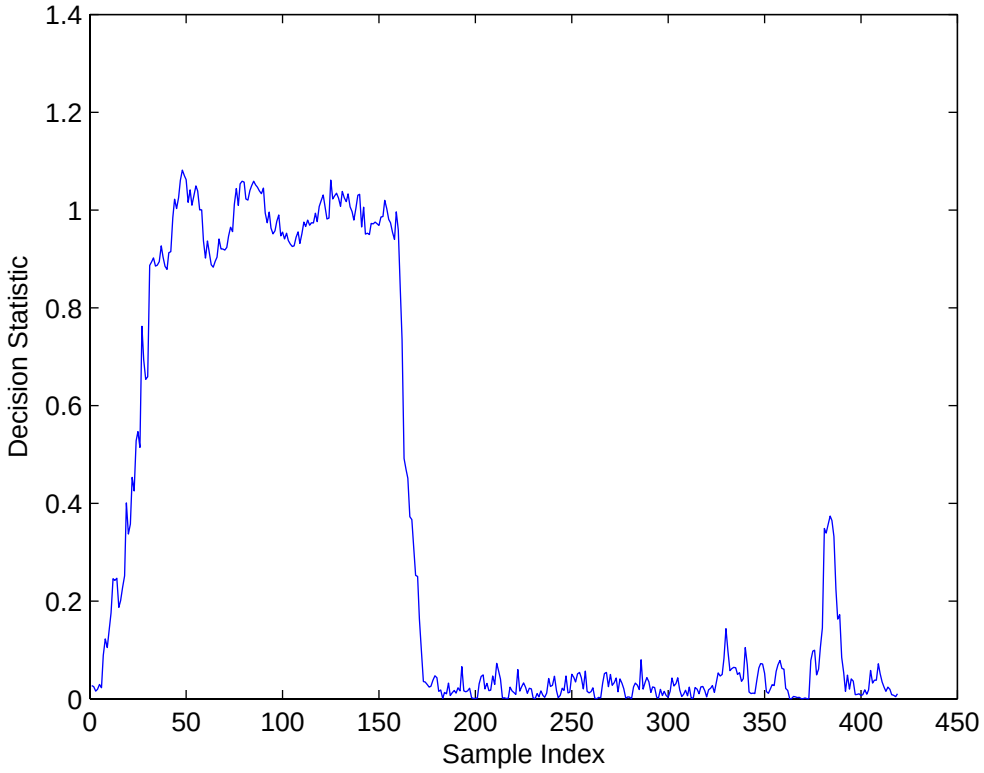


Figure 23. Autocorrelation Decision Statistic for IEEE 802.11a Short Preamble at 20 dB

To apply the autocorrelation method to IEEE 802.11a frame synchronization, the sample index at the end of the plateau, where the decision statistic drops below the threshold, needs to be found. With a correlator window of length sixteen, the drop off will be approximately 32 time samples before the end of the short preamble. In Figure 23, sample index 161 drops below this threshold. This would start the next symbol at sample index 193, which is one off from the actual symbol start at sample index 192.

Another frame synchronization technique is described in [8]. Here, the cross-correlation method is used and the algorithm searches for the peak outputs after the signal

has been detected. For IEEE 802.11a signals, there should be ten peaks that are sixteen time samples apart. When a peak is not found above the detection threshold after sixteen samples, the short preamble is assumed to have ended. This time index is taken as the beginning of the long preamble. For this method to succeed, all peaks must be above the threshold. If noise reduces one of the ten peaks, the system will assume the long preamble has started in the middle of the short preamble. The same methodology can be applied to the long preamble for frame synchronization by searching for the two peaks of the long preamble cross-correlation.

Figure 24 shows a comparison of the techniques described above for an IEEE 802.11a signal in a Rayleigh channel with rms delay spread of 5×10^{-8} s. Ten thousand runs were conducted over a SNR range from 0 to 30 dB. The decision threshold was 0.3. The transmitted signal consisted of three hundred complex samples followed by the IEEE 802.11a short and long preambles followed by 100 complex noise samples to simulate data. The offset from the ideal start of the first data sample was calculated for each run. The number of times the synchronization result landed on an index was counted and divided by the total number of runs to obtain the observed probability. The x-axis gives the offset from the time sample that is the actual start of the first data sample. The y-axis gives the average probability of the various techniques falling on that offset. Only those results falling within ±20 samples of the ideal sample index are shown. The figure clearly shows the superiority of using the cross-correlation technique. The results from the cross-correlation with the long and short preamble are statistically similar. The cross-correlation with the short preamble led to a sample offset of two samples from the ideal 94.5% of the time. The cross-correlation with the long preamble led to a sample offset of two samples from the ideal 92% of the time. The longer correlation window when using the long preamble makes this method more resistant to noise corruption. However, more peaks were counted using the short preamble. Thus, the repetition of the peaks from the ten symbols of the short preamble led to a slightly improved performance over the two peaks of the longer correlation window.

With the autocorrelation methods, the correlation window size does determine the effectiveness in tracking the end of the plateau. The drop off from the plateau is longer

with the long preamble, and thus the drop off is not as sharp. This accounts for the wider distribution of timing starts. The autocorrelation with the short preamble performs better in high SNR environments and its shorter length results in a steeper drop off as was seen in Figure 23. As the SNR decreases, the short autocorrelation does not perform as well due to the variance of the plateau height with increased noise.

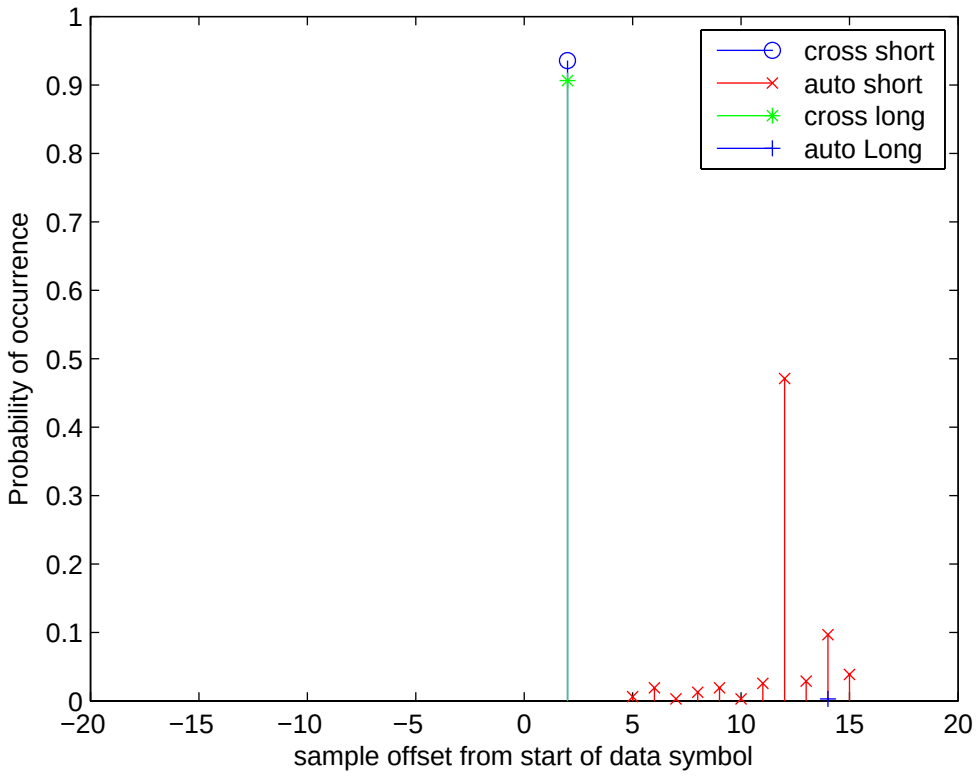


Figure 24. Time Synchronization Comparison of the Autocorrelation and Cross-Correlation Techniques in a Rayleigh Channel with RMS Delay Spread of 5×10^{-7} s Averaged over a Range of SNR values from 0 to 30 dB.

C. SUMMARY

This chapter discussed the format of the IEEE 802.11a and IEEE 802.16 preambles and their correlation properties. The cross-correlation values of the two preambles were found to be low in an ideal channel, suggesting that a single receiver

could be employed to search for both signals without a high rate of false alarm. The mean and variance of two signal detection methods were also compared. It was found that the decision statistic of the cross-correlation method had a higher average mean when the signal was present and a similar variance to the autocorrelation method. Furthermore, the cross-correlation decision statistic had a lower average mean when the signal was not present. For these reasons, the cross-correlation technique is a better technique to detect and classify IEEE 802.11a and IEEE 802.16 signals. Lastly, the decision statistics of both techniques were used for frame synchronization. It was found that the cross-correlation technique was found to have a consistently small timing offset in a given Rayleigh channel that was significantly smaller and more consistent than the autocorrelation technique. The following chapter will apply the cross-correlators developed in this chapter to detecting IEEE 802.11a, IEEE 802.16, and IEEE 802.11b signals in a Rayleigh channel to determine the efficacy of a single receiver for all three signals.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SIMULATION MODEL AND RESULTS

In Chapter III, the probability of detection and false alarm were analyzed for a correlator matched to one symbol of the IEEE 802.11a short preamble. In this chapter, a sampled signal will be received by a bank of correlators each matched to a different wireless networking preamble symbol. The objective is to determine the rates of detection and false alarm for each correlator when each signal of interest was present. The investigation centers on how well each correlator can filter out the other signals after being received in a Rayleigh channel. The three wireless networking signals used are from the IEEE 802.11a, IEEE 802.16, and IEEE 802.11b standards. These results are compared against a Gaussian noise input. The simulation model used to implement the detection and classification of the signals is described in detail followed by a discussion of the detection and frame synchronization results. The goal is to show that a single digital receiver with a pre-determined threshold can detect, classify and synchronize to multiple wireless networking signals.

A. SIMULATION MODEL

Figure 25 depicts a block diagram of the simulation model. The model generates an OFDM signal (IEEE 802.11a or IEEE 802.16), an IEEE 802.11b signal, or noise and sends that signal through a channel. The received signal was then applied to a bank of correlators matched to the different signals' preamble symbols as discussed in Chapter III. The normalized cross-correlation results were compared against a given decision threshold. A correlation output above the threshold value was considered a detection and classification of the signal data frame. Synchronization was then attempted for each of the signals. Following this, the physical layer frame header was demodulated to see if the transmitted information could be recovered. Each block of the model is described in detail below.

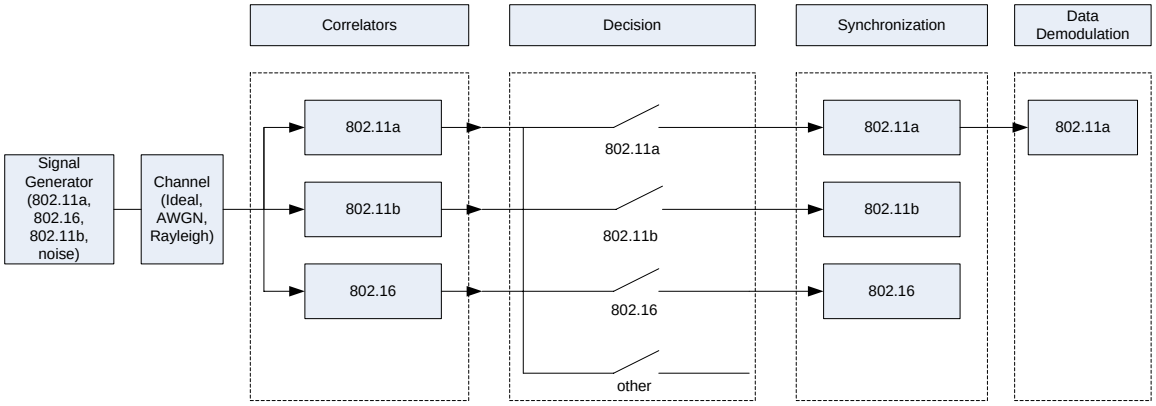


Figure 25. Simulation Set Up.

1. Signal Generation

The IEEE 802.11a, IEEE 802.16, and IEEE 802.11b signals generated consisted of the preamble and header bits. The IEEE 802.11a signals generated also included data symbols following the header information. The noise signal samples were complex with a Gaussian distribution with a mean of zero. The variance of the noise signal was set to 0.01 to match the power of the other signals. All the signals were then transmitted at the same power level. Further, all signals were transmitted and received at baseband. The following section will detail the process and structure of the transmitted signals. The baseband transmitter model for each type of signal included in each subsection includes in parenthesis the MATLAB M-files used to implement each function. These M-files are included in the Appendix for reference.

a. IEEE 802.11a

In the IEEE 802.11a standard, information is passed from the medium access control (MAC) layer in frames called MAC protocol data units (MPDU). The MPDU bits were generated using the randint.m function of MATLAB. Figure 26 depicts the implemented baseband block diagram for IEEE 802.11a signal generation.

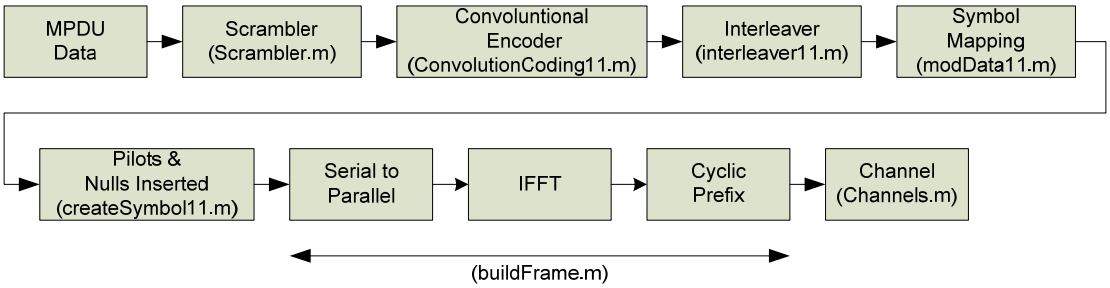


Figure 26. IEEE 802.11a Baseband Transmitter Model.

After the data bits are received from the MAC layer, they are first scrambled. The data bits are scrambled to prevent long strings of either ones or zeros being transmitted. Without scrambling, a string of 64 zeros or ones could cause the power of the transmission to be higher or lower than the average. This could cause the automatic gain control device in the receiver to operate in a non-linear region and cause distortion.

The scrambled bits are convolutionally encoded according to the data transmission rate as shown in Table 4. The base convolutional encoder is a rate $1/2$ encoder with a constraint length of seven that is punctured to obtain coding rates of $2/3$ and $3/4$. Convolutional forward error correction coding is used to detect and correct bit errors.

Following convolutional encoding, the bits are interleaved. Interleaving spreads adjacent encoded bits apart from each other. Convolutional decoders rely upon the surrounding bits to best determine if a bit was received in error. If there is a group of consecutive bit errors, the decoder will be unable to correctly decode the bits. When data bits are interleaved, their order is changed for transmission. If the channel induces a burst of consecutive errors, the bits in error will be spread apart at the receiver during the de-interleaving process. Bit errors are then more likely to be isolated and be correctable by the convolutional decoder.

Data rate (Mbps)	Modulation	Coding Rate	Coded bits per subcarrier	Coded bits per OFDM symbol	Data bits per OFDM symbol
6	BPSK	½	1	48	24
9	BPSK	¾	1	48	36
12	QPSK	½	2	96	48
18	QPSK	¾	2	96	72
24	16-QAM	½	4	192	96
36	16-QAM	¾	4	192	144
48	64-QAM	2/3	6	288	192
54	64-QAM	3/4	6	288	216

Table 4. Rate Dependent Parameters (From Ref [1]).

The interleaved bits are mapped into symbols dependent upon the data rate according to Table 4. Higher-order symbol mapping allows more than one data bit to be represented by a signal symbol and, thus, allows for a faster transmission rate. Higher-order symbol mappings are more susceptible to noise interference and more power is required to keep the bit error rate constant as the number of bits per symbol increases for a constant bandwidth.

Following interleaving, the bits are mapped into symbols according to Table 4. The modulation techniques used are BPSK, QPSK, 16-QAM, or 64-QAM. After this, four pilot tones are inserted to aid with frequency tracking. In addition, twelve nulls are added as discussed earlier. The resulting 64 values are transformed into time domain samples by a 64-point IFFT. Sixteen cyclic prefix samples are prepended and the resulting 80 time domain samples are sent to the channel.

b. IEEE 802.16

Figure 27 depicts the block diagram for IEEE 802.16 signal generation. The process is similar to IEEE 802.11a signal generation. IEEE 802.16 signals are generated differently depending upon whether they are on the uplink or the downlink. The standard also allows for several optional aspects to physical layer signal generation. The intent here was to create a basic IEEE 802.16 signal. A downlink signal was selected with a cyclic prefix extension of 64 samples. The signal created had 256 subcarriers of which 192 were used to carry data. The IEEE 802.16 standard uses the term randomizer

as opposed to scrambler for the process to jumble the data bits to prevent long string of ones or zeros being transmitted. Next, IEEE 802.16 employs a concatenated forward error code correction. The two concatenated codes are a Reed-Solomon code and a binary convolutional code. A shortened Reed-Solomon technique is employed first. The code is shortened depending upon the data rate for transmission. To perform Reed-Solomon coding, the input is first converted to a Galois field (2^8). The encoder accepts 239 bytes of data and returns 255 bytes of output data. When the code is shortened for lower data rates, zeros are inserted before encoding; the same number of zeros are removed after encoding.

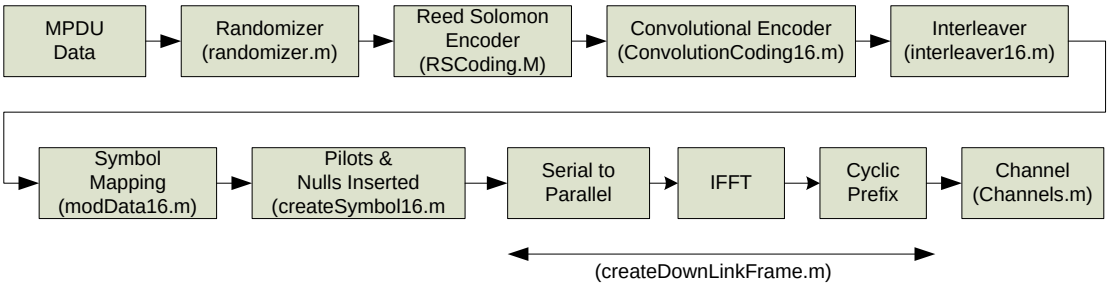


Figure 27. IEEE 802.16 Baseband Transmitter Model.

Following the Reed-Solomon encoder is a 1/2 rate binary, convolutional encoder with a constraint length of 7. The code is punctured depending upon the modulation used. The data bits are then interleaved and mapped into BPSK, QPSK, 16-QAM, or 64-QAM symbols. Pilots and nulls are added before a 256-point IFFT is taken. The cyclic prefix of the last 64 samples of the symbol is added to the front of the symbol before transmission.

c. IEEE 802.11b

The IEEE 802.11b signal can transmit data at rates of 1, 2, 5.5, or 11 Mbps [2]. The 1- and 2-Mbps signals are transmitted using direct sequence spread spectrum with a differential modulation of either BPSK or QPSK. The 5.5 and 11 Mbps signals are transmitted using eight bit-complementary code keying. The passband signals are transmitted at 2.4 GHz. The standard allows for two different preamble and header

formats. One is backwards compatible with the earlier version of the standard and another is only compatible with systems capable of transmitting at the higher data rates of 5.5 and 11 Mbps. The former is only transmitted at data rates of 1 and 2 Mbps. A sixteen bit CRC, used to check for errors in the physical layer convergence procedure (PLCP) header, is added as the last part of the PLCP header. Figure 28 depicts the functional block diagram of the IEEE 802.11b signal generation used in the experiment. The simulated signal was a 1 Mbps, differential BPSK waveform with the backwards compatible preamble and PLCP header.

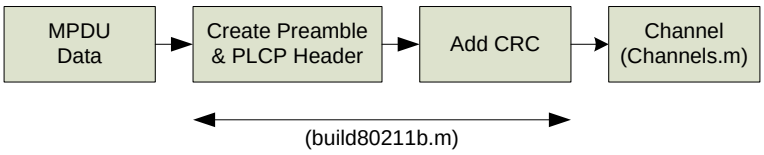


Figure 28. IEEE 802.11b Baseband Transmitter Model.

2. Channel

The channels included in the model were ideal, AWGN, and multipath. For the AWGN channel model, the only input required was a SNR. Multipath channels are typically characterized as either Ricean or Rayleigh. A Ricean channel is a multipath channel that includes a line of sight path from the transmitter to the receiver. The Rayleigh channel does not include a direct line of sight path. All received paths are reflections. Thus, the Rayleigh channel can typically cause more signal distortion and a lower received signal power than a Ricean channel. A Rayleigh channel was chosen as the multipath channel for the simulations.

Rayleigh channels are characterized by their rms delay spread as was discussed in Chapter II. Typical indoor rms delay spreads for an in-door environment range between 10^{-9} to 10^{-7} seconds [10]. Since OFDM signals were designed to handle delay spreads less than the cyclic prefix length (0.8 μs for IEEE 802.11a), a rms delay spread of 5×10^{-8} seconds was chosen. This rms delay spread was applied over a range of SNRs to test the detection and synchronization algorithms in a range of channel environments.

The simulations were first run with no Doppler spread in the channel. Following this, a simulation was conducted with a Doppler spread of 20 Hz to show the degradation effects of Doppler spread on OFDM symbols.

3. Detection

The detection algorithm employed was the cross-correlation technique discussed in Chapter III. Each input was cross-correlated with the IEEE 802.11a, IEEE 802.16, and IEEE 802.11b preambles. The results were compared against a predetermined decision threshold. The simulation was conducted twice: first, using a decision threshold of 0.3 and then a decision threshold of 0.6. These two values were chosen to show the effects on P_d and P_f of the decision threshold. If the value was above the threshold, detection occurred. In this way, one received signal could cause multiple detections.

For IEEE 802.11a signals, the cross-correlation was applied to the short preamble and the long preamble was used for channel estimation as intended by the standard. The cross-correlation output was normalized by the preamble power. The detection algorithm searched for all ten peaks within the short preamble. Only if all ten peaks were found at the correct spacing of sixteen samples apart was a signal detected. In this way, the probability of false alarm was considerably lower than discussed in Chapter III. The probability of false alarm rates discussed in Chapter III were based on using only one peak of the cross correlation output for detection. The probability of having ten values above the threshold spaced exactly sixteen samples apart is ten times lower.

For IEEE 802.16 signals, the cross-correlation technique was applied to the first preamble and the second preamble was used for channel estimation in the same approach as was used for the IEEE 802.11a signals. The signal detection algorithm searched for all five peaks of the first preamble in order to detect and classify the signal.

For IEEE 802.11b signals, the detection algorithm cross-correlated the received signal with the 144 bit PLCP preamble. This value was normalized by the power of the PLCP preamble. A value over the threshold was considered a detection and a classification of the signal.

4. Synchronization

For each of the signals, frame synchronization began after detection was made. For the IEEE 802.11a signal, synchronization started from the sample index of the tenth peak of the detection algorithm. This peak marked the end of the short preamble. The next 160 received samples were the long preamble and the first sample after that was the start of the header.

In a similar manner, the start of the IEEE 802.16 data was found by adding 64 to the sample index of the last peak of the cross-correlation and then adding another 320 to get to the end of the second preamble and the start of the header.

For IEEE 802.11b, the peak of the cross-correlation occurred at the end of the SYNCH field of the preamble as shown in Figure 29. Sixteen bits were added to this index to arrive at the first bit of the PLCP header. The signal and length fields can then be extracted and the CRC bits checked. After adding 48 to the start of the PLCP header index, the starting index of the data bits is found.

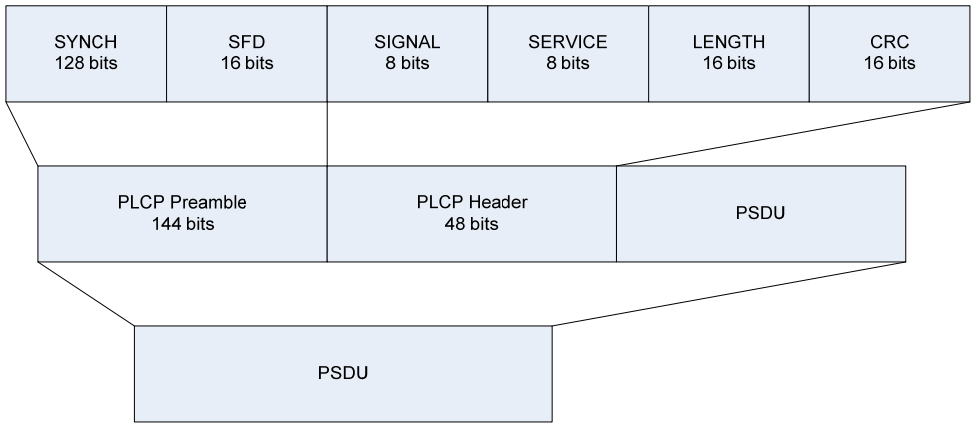


Figure 29. Long PLCP PPDU Format (from Ref [2]).

5. Demodulation

For IEEE 802.11a signals that were detected and classified, demodulation was attempted after frame synchronization. The first step was to correct for phase distortion

caused by the channel. A time-domain, channel estimation was employed. Since the channel acts as a filter, the received signal samples can be written in discrete time as

$$x[n] = h[n] \otimes s[n] + n[n] \tag{3.29}$$

where $\otimes$ denotes circular convolution due to the circular nature of the transmitted signal because of its cyclic prefix and $s[n]$ is the transmitted signal. Equation (4.1) can be re-written in matrix notation as

$$\mathbf{x} = \mathbf{S}\mathbf{h} + \mathbf{n} \tag{3.30}$$

or

$$\begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[N-1] \end{bmatrix} = \begin{bmatrix} s[1] & s[N-1] & \cdots & s[N-M+1] \\ s[1] & s[0] & \ddots & s[N-M+2] \\ \vdots & \vdots & \ddots & \vdots \\ s[N-1] & s[N-2] & \cdots & s[N-M] \end{bmatrix} \begin{bmatrix} h[0] \\ h[1] \\ \vdots \\ h[N-1] \end{bmatrix} + \begin{bmatrix} n[0] \\ n[1] \\ \vdots \\ n[N-1] \end{bmatrix} \tag{3.31}$$

where M is the length of the impulse response of the channel. The value of M needs to be at least sixteen since that is the number of samples in the IEEE 802.11a cyclic prefix. A higher value will increase the accuracy of the estimate while taking more time to process. The value of M used was sixteen.

For an estimate of the impulse response of the channel to be obtained, the transmitted signal must be known at the receiver. This is accomplished with the long preamble. After synchronization, the receiver knows which received samples represent the long preamble. By inserting the known long preamble samples for the sent signal in (4.2), the equation can be solved for the impulse response of the channel. This can be accomplished through the pseudo inverse approach:

$$\mathbf{S}^{+}\mathbf{x} = \mathbf{h} + \mathbf{S}^{+}\mathbf{n} \tag{3.32}$$

where

$$\mathbf{S}^{+} = (\mathbf{S}^{T}\mathbf{S})^{-1}\mathbf{S}^{T} \tag{3.33}$$

is the pseudo inverse. Since the noise samples are assumed to be zero mean, the expected value of the product of the noise samples with the pseudo inverse will be zero, leaving the estimate of the channel impulse response to be:

$$\mathbf{h} = \mathbf{S}^{+}\mathbf{x}\,. \tag{3.34}$$

A 64-point FFT was then taken to get the frequency response of the channel as shown in Figure 30 for a Rayleigh channel with an rms delay spread of 0.5×10^{-7} seconds. Instead of the flat channel response of the ideal channel, this channel has a deep fade in the beginning and the middle of the frequency band.

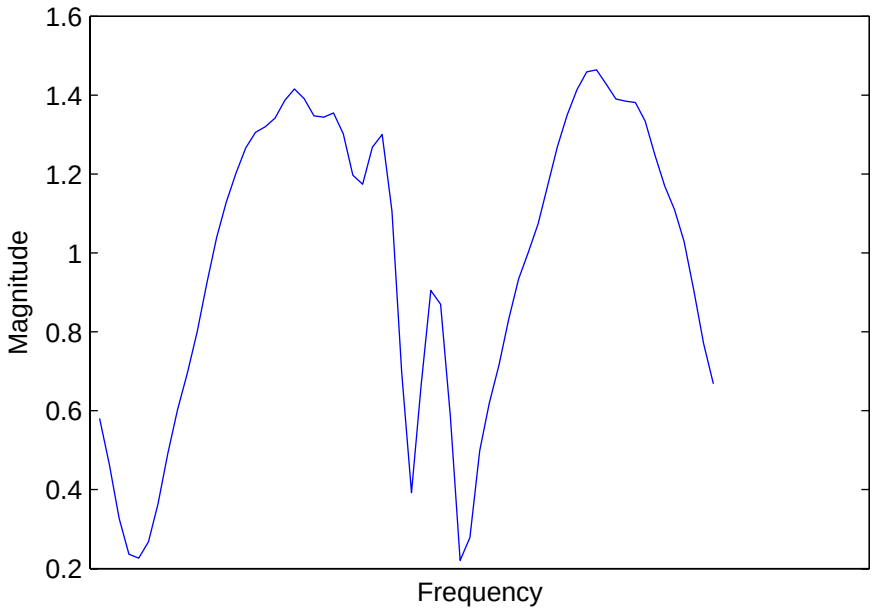
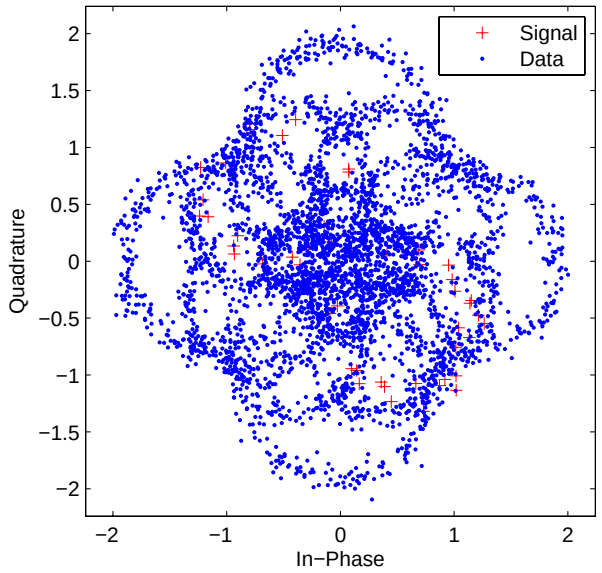
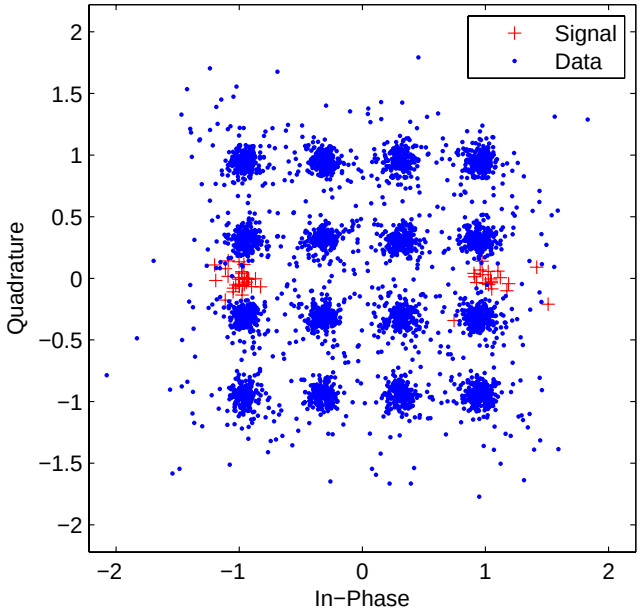


Figure 30. Channel Frequency Response for Rayleigh Channel with RMS Delay Spread of 5×10^{-7} s and a SNR of 20 dB.

After the signal data has been brought back into the frequency domain, the channel correction can be applied. Since convolution in the time domain is multiplication in the frequency domain, the estimate of the received signal is obtained by dividing the received signal by the channel frequency response. Figure 31 shows a received 16-QAM signal constellation from a Rayleigh channel and the signal constellation after the channel estimation has been applied. The pluses are the location of the BPSK signal field samples and the dots are the 16-QAM data samples.



(a)



(b)

Figure 31. Channel Estimation: (a) Uncorrected and (b) Corrected 16-QAM Signal from a Rayleigh Channel with RMS Delay Spread of 5×10^{-8} s and a SNR of 20 dB.

Following channel estimation, the demodulation process is the same as the transmission process described above except in reverse. Figure 32 presents the block

diagram of the IEEE 802.11a receiver. The constraint length of the convolutional decoder was three and hard decision decoding was employed. Coding gain will be seen if the constraint length is extended or if soft decision decoding is applied.

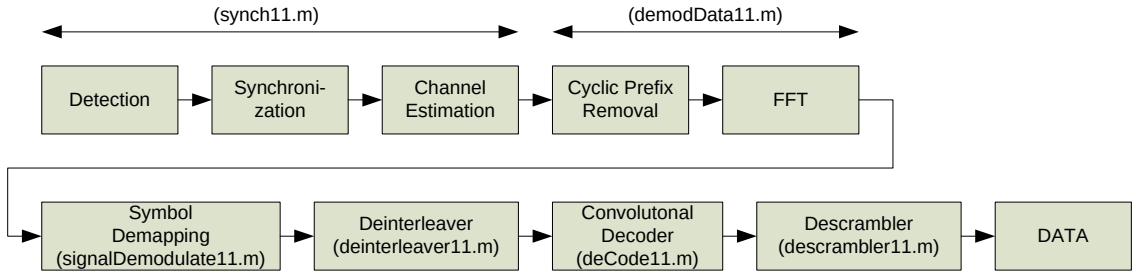


Figure 32. IEEE 802.11a Baseband Receiver Block Diagram.

Once the signal had been demodulated, the bits were parsed according to the standard to read the PLCP header. Figure 33 gives the format of the IEEE 802.11a physical layer frame. The SIGNAL field, along with the 16 service bits, comprises the PLCP header. The SIGNAL field informs the receiver of the parameters needed to decode the following data symbols. It contains the length in bytes of the data sent from the MAC layer to the physical layer. The length, which can be from 1 to 4095 bytes, determines the number of OFDM data symbols to follow and allows the receiver to know how many symbols should be received. The SIGNAL field also contains the data rate at which all of the following symbols in the physical layer frame will be transmitted. The SIGNAL field has a parity bit for basic error checking, in addition to the forward error code correction that is applied to all OFDM symbols. The SIGNAL field is always transmitted with BPSK modulation and rate ½ convolutional coding for a transmission data rate of 6 Mbps. The OFDM symbol containing the SIGNAL field is transmitted at the system's most robust data rate to best ensure correct reception. If there is a bit error in the data rate field or the length field, the following OFDM symbols will not be decoded properly. At the end of the SIGNAL field, seven bits are added to initialize the receiver's scrambler, and nine bits are reserved for future use. These bits are termed the service bits. The six tail bits added at the end are to return the convolutional encoder to the zero state. Pad bits are added, when necessary, to insure there are enough bits to fill an integer number of OFDM symbols.

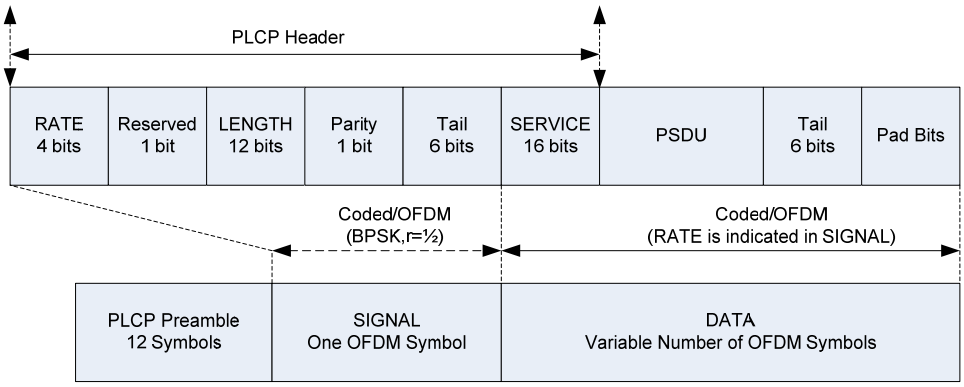


Figure 33. PPDU Frame Format (From Ref [1]).

The IEEE 802.16 signal is demodulated in a similar manner to the IEEE 802.11a signal. Figure 34 depicts the block diagram for the implemented IEEE 802.16 signal demodulation. The first significant difference is that a Reed-Solomon decoder is employed following the binary convolutional decoder. The second significant difference is that a 256-point FFT is taken to bring the time domain samples back into the frequency domain for processing.

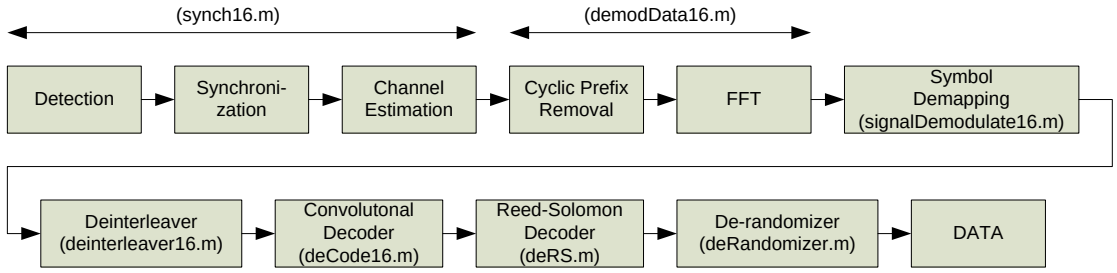


Figure 34. IEEE 802.16 Baseband Receiver Block Diagram.

B. RESULTS

This section presents the results from the simulation. The focus of the results is the probability of detection of the intended signal for each correlator and the probability of false alarm for the same correlator caused by the other signals. Frame synchronization results will also be discussed along with the probability of demodulating an OFDM frame header when the entire OFDM frame cannot be demodulated due to noise.

1. IEEE 802.11a

An IEEE 802.11a signal was generated and sent through a Rayleigh channel with a rms delay spread of **Error! Objects cannot be created from editing field codes.**s. This rms delay spread is shorter than the cyclic prefix length of **Error! Objects cannot be created from editing field codes.**and so ISI should not degrade reception. As discussed earlier, a value longer than this would cause ISI. The results with ISI would not be easily comparable due to the random nature of the ISI. A value was chosen to show the system operating in a significant multipath environment. The detection threshold was set to 0.3 to show the capability of the system in a low SNR environment. The resulting probability of detection of each of the correlators is shown in Figure 35 where 10,000 packets were transmitted at each SNR value from 0 to 15 dB. The size of the data portion of the packet was 100 bytes.

As can be seen in Figure 35, even at 0 dB in a Rayleigh channel, there were no false detections by either the IEEE 802.11b or 802.l6 filters. By a SNR of 13 dB, the probability of detection was 100%. Thus, a digital receiver can be used to distinguish IEEE 802.11a signals from both IEEE 802.16 and IEEE 802.11b signals. Furthermore, the IEEE 802.11a signals achieved 50% detection at a SNR of between 3 and 4 dB.

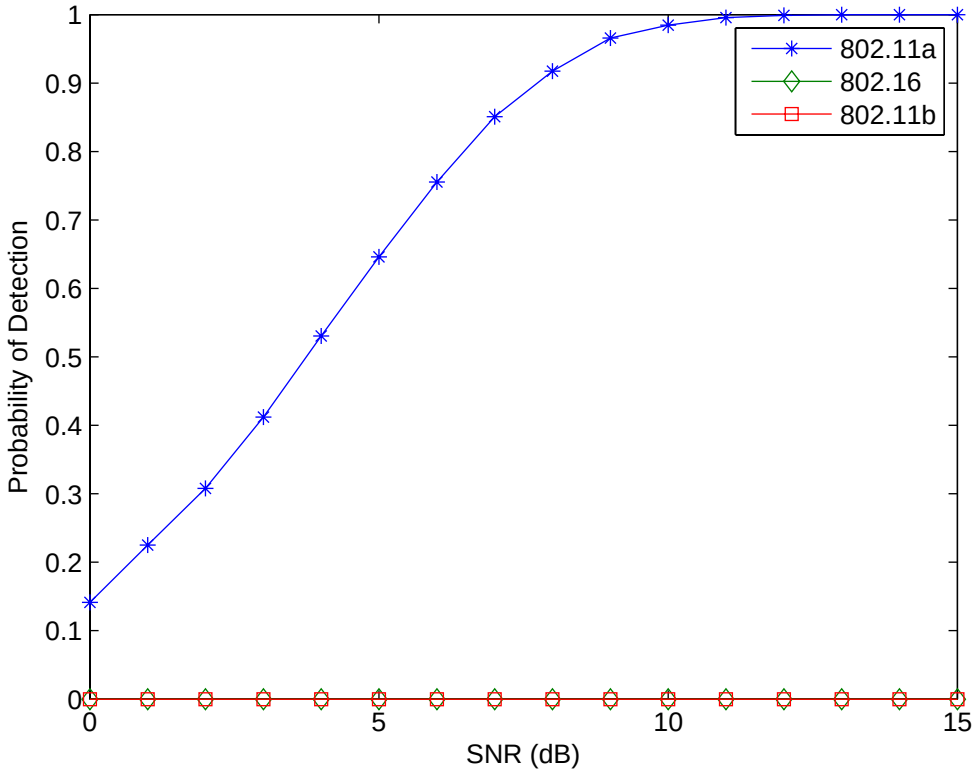


Figure 35. Probability of Detection in a Rayleigh Channel with an RMS Delay Spread of 5×10^{-8} s and a detection threshold of 0.3.

Figure 36 shows the results of the parity bit check of the same IEEE 802.11a signals as in Figure 35 where 10,000 packets were transmitted at each SNR value from 0 to 15 dB. The graph shows the probability of a correct parity bit check and the probability of detection. The parity bit cannot be computed unless the signal is detected. Thus, the probability of a good parity bit is always lower than the probability of detection. Figure 36 shows that, if the packet is detected, the probability is high that the parity bit check will pass. This indicates that if the signal is detected by the system, the probability of bit error in the signal field will be low. Only at low SNRs does the probability of a good parity bit check not match the probability of detection. The packet header performs well at low SNR values because it is transmitted using BPSK modulation with rate ½ coding.

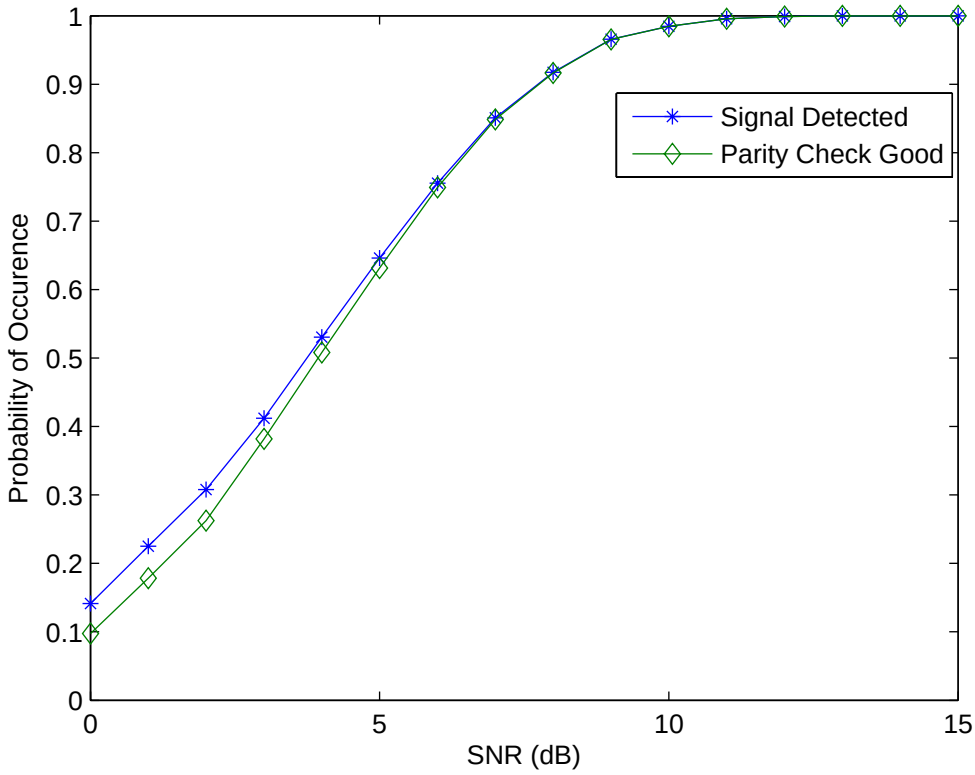


Figure 36. Probability of Correct Parity Bit in a Rayleigh Channel with RMS Delay Spread of 5×10^{-8} s and a Detection Threshold of 0.3.

Typically, if the parity bit check detects an error, the frame will be discarded. However, useful information can still be gathered even if the parity bit check fails. The data rate and data length fields of the header can still be checked to see if the bit errors may have left them unaffected. Table 5 shows the probability of a correct parity bit check and the average bit error rate of the header for all packets detected, regardless of the parity bit check result. At a SNR of 6 dB, 20% of the packets had a bad parity bit check, but the average header BER was only 0.0029. Thus, most likely, few bit errors occurred per packet header. Furthermore, a bit error in the tail subsection or the reserved subsection of the header, which account for 7 of the 24 bits in the header, would not affect the information of interest. The conclusion is that even when the parity bit check is bad and a typical system would discard the packet, there may still be information available regarding the packet.

Table 5 also shows the average BER for the data transmitted following the header. The synchronization result is also shown as an offset from the ideal start of the first data symbol sample. To calculate the offset, the synchronization result of all detected packets was used. The multipath fading causes a near-constant delay of two samples from the actual start of the data symbol. The error caused by this delay is small enough to be compensated for by the error correction code. The BER of the data, which was transmitted at 12 Mbps using QPSK modulation, decreased as the SNR increased. Since the data size within the packet was 800 bits, all packets fell well within the coherence time of the channel with a Doppler spread of 50 Hz. The channel coherence time according to (2.11) is 8.46 ms. The total packet transmission time is

$$8\ \mu s + 8\ \mu s + \frac{48 \text{ bits}}{6\times10^6 \text{ b/s}} + \frac{800(2) \text{ bits}}{12\times10^6 \text{ b/s}} = 158\ \mu s << 8.46 \text{ ms} \tag{3.35}$$

where the first two terms are the time for the short and long preambles, the third term is time to transmit the packet header symbol, and the last term is the time to transmit the data after being encoded at rate ½. The velocity associated with 50 Hz of Doppler spread is 3 m/s. This is applicable for an office environment where objects in the channel may be moving, but not applicable for a mobile environment. From (4.5) and Table 5, the packets were not significantly affected by the Doppler spread of the channel. Lastly, it is important to note that the middle four columns of results in Table 5 do not change with a change in the data modulation because the signal field is always transmitted using BPSK modulation.

SNR	Probability of Detection	Probability of Good Parity Check	Avg offset from Ideal Start	Average Header BER	Average Data BER
0	0.1413	0.0974	1.9887	0.1329	0.4733
1	0.2252	0.1784	1.9929	0.0781	0.4287
2	0.3078	0.2624	2	0.0458	0.354
3	0.4119	0.3819	2	0.0238	0.251
4	0.5305	0.508	2	0.0119	0.161
5	0.6461	0.6313	2	0.0061	0.0948
6	0.7555	0.7494	2	0.0029	0.0535
7	0.8509	0.8487	2	0.001	0.0295
8	0.9173	0.9165	2	0.0004	0.0164
9	0.9656	0.9655	2	0.0001	0.0089
10	0.9845	0.9845	2	0	0.005
11	0.9957	0.9957	2	0	0.0027

SNR	Probability of Detection	Probability of Good Parity Check	Avg offset from Ideal Start	Average Header BER	Average Data BER
12	0.999	0.999	2	0	0.0014
13	0.9998	0.9998	2	0	0.0007
14	1	1	2	0	0.0003
15	1	1	2	0	0.0002

Table 5. IEEE 802.11a Frame Demodulation Results in a Rayleigh Channel with RMS Delay spread of 5×10^{-7} s and Detection Threshold = 0.3.

To further illustrate the information available in the packet header when the parity check fails, Table 6 shows the results of the received data rate field from the packet header when the parity check was good and when the parity check was bad for signals that were detected. The total number of times out of the 150,000 packets that the parity check was good, when the signal was detected, was 114,737. The total number of times that the parity check was bad, when the signal was detected, was 2,216. The table shows that even with a bad parity check the data rate information in the packet header was still received correctly over 50 % of the time and that the percentage that the misreading was something other than zero was only 3.85 %. The data rate field is not sent in its equivalent binary value but as a binary code to help prevent single bit errors from causing a bad reception.

Received Data Rate	Probability of Data Rate Value When Parity Check Was Good	Probability of Data Rate Value When Bad Parity Check Was Bad
No match	0.25 %	37.7 %
6	0.000878 %	0.517 %
9	0.0451 %	0.517 %
12	99.6 %	58.3 %
18	0.0 %	2.16 %
24	0.0 %	0.0 %
36	0.00555 %	0.0 %
48	0.000435 %	0.658 %
54	0.0000868 %	0.0 %

Table 6. Received Data Rate Value When the Packet Header Parity Check Failed and the Actual Data Rate Value Sent Was 12.

The data length field was received correctly 76.49 % of the time when the signal was detected. Within this, the data length field was received correctly, 98.93 % when the

parity bit was correct and 36.64 % when the parity bit was incorrect. The data length field is sent in its binary equivalent form and, thus, is more susceptible to bit errors than the data rate field.

When the data rate and length fields of the preamble of the packet are received correctly, all other information needed for demodulation of the packet is available. From to Table 4, the coding rate and modulation of the data symbols can be deduced from the data rate. The number of symbols in the packet can be calculated from the data rate and the length fields according to the equation

$$N_{SYM} = \lceil (16 + 8 \times L_{MAC} + 6) / N_{DBPS} \rceil \tag{3.36}$$

where N_{DBPS} is defined in Table 4 for a given data rate, L_{MAC} is the length of the data in bytes from the MAC layer, and $\lceil \quad \rceil$ is the ceiling function.

In summary, IEEE 802.11a signals can be successfully distinguished from IEEE 802.11b and IEEE 802.16 signals with no concern for processing time being wasted on false detections in the 802.16 or 802.11b detectors. Furthermore, in low SNR environments, significant packet header may be available even when the full packet cannot be demodulated.

2. IEEE 802.11b

Figure 37 shows the results when an IEEE 802.11b signal was received. The signal was transmitted 10,000 times for each SNR value from 0 to 30 dB. The length of the data in each packet was 100 bytes. There were no false detections by either the 802.11a or 802.16 detectors. The rate of detection by the 802.11b correlator is significantly higher because the correlation window is 128 samples long. However, the resulting CRC failed nearly 100% of the time up to 5 dB of SNR. By 10 dB of SNR, the probability of detection is 100%, but the probability of a correct CRC is 50%. This compares to the IEEE 802.11a results where at 10 dB of SNR the signal detection and probability of parity bit check were both over 90%. The probability of good CRC check for the IEEE 802.11b signals is lower for two reasons. First, the 802.11a signal has error

correction coding to catch and correct errors, where the 802.11b signal does not. Secondly, the CRC is a more powerful algorithm for detecting bit errors than a parity bit check.

From Table 7, the synchronization algorithm consistently chose the sample two after the ideal for the start of the first sample of the header. Further, from a signal detection standpoint, over 98% of the packets received with a SNR of 7 dB were detected, but only 11.3% of those packets had a successful CRC while the packet header BER was only 6%. Thus, for IEEE 802.11b signals, there is also significant opportunity to demodulate the frame header even when the rest of the packet cannot be demodulated.

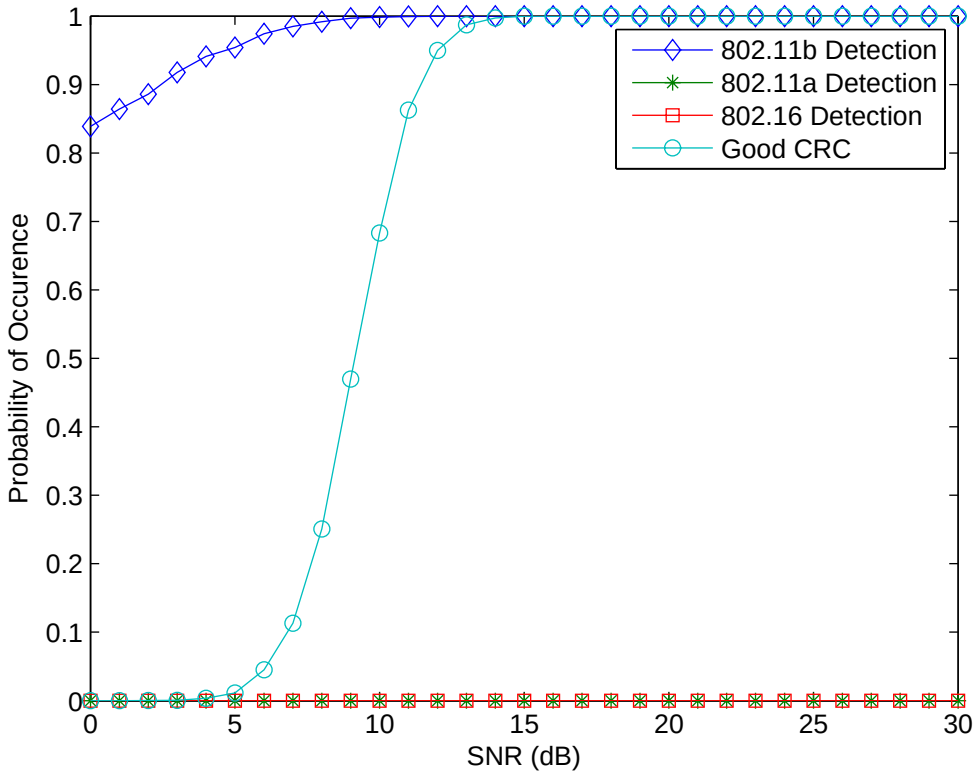


Figure 37. Probability of IEEE 802.11b Signal Detection and Correct Parity Bit Check in a Rayleigh Channel with RMS Delay Spread of 5×10^{-8} s.

SNR	Probability of Detection	Probability of Good CRC	Avg. Sample Offset from Start of data	Average Header BER	Average Data BER
0	0.8387	0	2	0.2835	0.5
1	0.8643	0	2	0.2518	0.4998
2	0.8859	0.0001	2	0.2205	0.5004
3	0.9178	0.0004	2	0.1843	0.5
4	0.9412	0.0034	2	0.1512	0.4988
5	0.9542	0.0111	2	0.1194	0.4951
6	0.9741	0.045	2	0.0868	0.4796
7	0.9849	0.113	2	0.0608	0.4475
8	0.9917	0.2507	2	0.0383	0.3811
9	0.997	0.4699	2	0.0212	0.272
10	0.9984	0.683	2	0.0105	0.1639
11	0.9995	0.8625	2	0.004	0.0722
12	0.9999	0.9499	2	0.0014	0.0272
13	1	0.9871	2	0.0004	0.0079
14	1	0.9973	2	0.0001	0.0026
15	1	0.9999	2	.0000708	0.0013
16	1	1	2	.000002083	0.0013
17	1	1	2	0	0.0012
18	1	1	2	0	0.0012
19	1	1	2	0	0.0013
20	1	1	2	0	0.0012

Table 7. IEEE 802.11b Signal Detection and Demodulation Results in a Rayleigh Channel.

3. IEEE 802.16

An IEEE 802.16 signal was then run through the detector under the same conditions as the IEEE 802.11a signals. The probability of detection results are plotted in Figure 38. There were no false detections in either the 802.11a or 802.11b detectors. The probability of detection for 802.16 signals is significantly higher at lower SNR levels than with the 802.11 detector due to the longer correlation window. The frame synchronization technique consistently chose the data sample two after the ideal start of the header.

The channel estimation function for the IEEE 802.16 detection was not implemented in this work. This does not impact the signal detection results but does impact the cyclic redundancy check results. The probability of good CRC plotted in Figure 38 is the worst case performance, and there would be significant improvement if channel estimation was implemented. The probability of CRC shown here is better than

that for IEEE 802.11b because the IEEE 802.16 standard implements rate ½ convolutional coding for the packet header where as the IEEE 802.11b standard does not utilize forward error correction coding. Reed-Solomon coding is not implemented for the packet header in IEEE 802.16. The IEEE 802.16 standard takes the best from the two other standards in that it implements convolutional coding and then applies an eight-bit CRC for additional error detection. This further reduces the receiver's attempts to demodulate packets where there has been a bit error in the packet header that could cause the rest of the packet to be demodulated incorrectly.

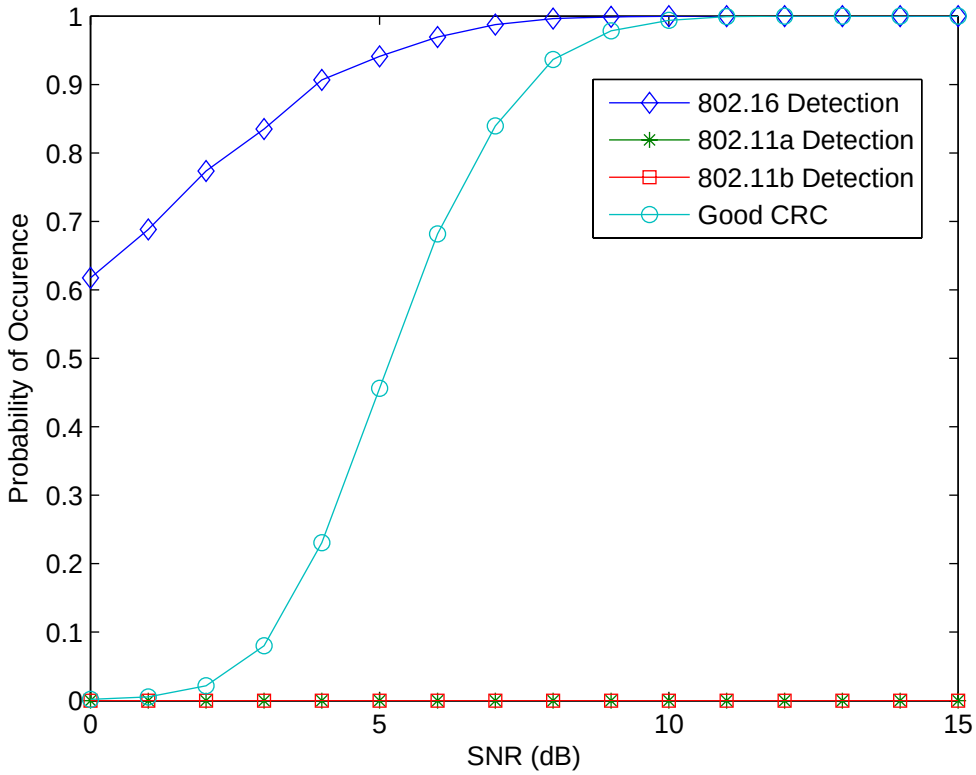


Figure 38. Probability of IEEE 802.16 Signal Detection and Correct Parity Bit Check in a Rayleigh Channel with RMS Delay Spread of 5×10^{-8} s.

4. Noise

To test the system's probability of false alarm, a white noise interference signal was transmitted with a signal power 5 dB lower than the data signals. This signal was intended to act as a random interference signal in the frequency band of interest. The

noise signal was a stream of 10,000 complex samples that was run 100 times over a SNR range of 0 to 20 dB. With a threshold of 0.3, no false alarms were detected by either the 802.11a, 802.11b, or 802.16 detectors. These results are significantly better than the theoretical predictions for the autocorrelation method with a window length of sixteen presented in Chapter III. The theory predicted that autocorrelation with a length-sixteen window with a threshold of 0.3 would result in false alarm rate of almost 80%. However, the cross-correlation technique coupled with the search for multiple, consecutive peaks had an observed P_f of less than 10^{-4}. For autocorrelation with a window length of 64, the theory predicted a P_f of 5×10^{-4}. The observed cross-correlation with the IEEE 802.16 short preamble resulted in a P_f less than 10^{-4}. The P_f for autocorrelation with a window length of 128, as in the case of IEEE 802.11b signals, is 10^{-8} and enough samples were not run to test if the cross-correlation could match that performance.

5. Summary

By looking at the detection probabilities at 0 dB for each of the signals, the impact of the correlation window is apparent. As the correlation window size increases from 16 to 64 to 128, the probability of signal detection increases from 0.1, 0.6, and 0.8. Better signal detection results could then be obtained for the IEEE 802.11a and IEEE 802.16 signals by correlating the entire short or first preamble. For the IEEE 802.11a case, the window length would be 160 samples long. This would not aid in signal demodulation but should produce significantly better signal detection results.

Table 8 through Table 10 summarize the probability of detection results for the baseband digital receiver presented in this chapter. The three tables give the probabilities of detection for each signal by all of the detectors for SNR values of 3, 7, and 15 dB, respectively.

Signal Transmitted	Probability of Detection		
	802.11a	802.11b	802.16
802.11a	0.4119	0.0	0.0
802.11b	0.0	0.9178	0.0
802.16	0.0	0.0	0.8351

Table 8. Probability of Detection of IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 Signals by a Digital Receiver with a SNR of 3 dB.

Signal Transmitted	Probability of Detection		
	802.11a	802.11b	802.16
802.11a	0.8509	0.0	0.0
802.11b	0.0	0.9849	0.0
802.16	0.0	0.0	0.9874

Table 9. Probability of Detection of IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 Signals by a Digital Receiver with a SNR of 7 dB.

Signal Transmitted	Probability of Detection		
	802.11a	802.11b	802.16
802.11a	0.9957	0.0	0.0
802.11b	0.0	0.9999	0.0
802.16	0.0	0.0	1.0

Table 10. Probability of Detection of IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 Signals by a Digital Receiver with a SNR of 15 dB.

The most significant fact is that no false detections occurred by any of the unintended detectors. This means that a single, baseband, digitial receiver can be employed to detect and demodulate IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 signals with no significant false alarm rates from either noise signals or an unintended detector.

The frame synchronization method was also shown to work consistently for all three signals. Table 11 shows the percentage of detected signals where the frame synchronization technique chose as the first sample of the packet header a sample that was two sample periods after the actual first sample of the packet header. For this channel, a simple subtraction of two applied to the synchronization result would have led to almost perfect frame synchronization over a wide range of SNR values. Further study could be conducted to determine, over a range of multipath channel conditions, an average frame offset that could be applied to improve bit error rates in a wider range of channel conditions.

Signal	% frame starts two samples after the ideal start
IEEE 802.11a	99.998 %
IEEE 802.16	100 %
IEEE 802.11b	100 %

Table 11. Frame Offset Results for all Three Signals in a Rayleigh Channel.

This chapter presented the simulation model used to analyze the efficacy of a digital receiver to detect multiple wireless networking signals. The simulation results indicate that a single, digital receiver can be used to detect, classify and frame synchronize to multiple wireless networking signals with an acceptable rate of false alarm from unintended signals.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS

The objective of this thesis was to investigate if a single, digital, baseband receiver could detect, classify, and synchronize to IEEE 802.11a, IEEE 802.16, and IEEE 802.11b signals with minimal rates of false alarm and high rates of detection. The packet information available when an IEEE 802.11a signal was detected, but could not be fully demodulated, was also investigated.

A. SUMMARY OF THE WORK DONE

First, a simulation model for the generation and demodulation of IEEE 802.11a OFDM signals was implemented in MATLAB. Second, a partial model for the generation and demodulation of IEEE 802.116 OFDM signals was developed in MATLAB. Third, a signal detection and synchronization model was developed to test the cross-correlation results of each signal by the other detector. Additionally, an attempt was made to define the probability density functions of the decision statistic at the receiver when a signal of interest was present and when only noise was present. From this result, a ROC curve was determined to select thresholds based upon desired probabilities of detection and false alarm. Lastly, the signal generation, detection, demodulation models were put together with a Rayleigh channel model to gather detection and synchronization results.

B. SIGNIFICANT RESULTS

A single, digital, baseband receiver can separately detect and classify IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 signals from each other. A bank of correlators matched to a symbol of each signal's preamble was shown to be a successful implementation for distinguishing IEEE 802.11a, IEEE 802.11b, and IEEE 802.16 signals from each other with negligible probabilities of false alarm by the unintended detector. The result of this is that a single digital receiver attached to an air interface capable of receiving and sampling all three signals can detect and demodulate each signal

without concern for wasting processing power or missing signals due to false detections. Further, the results showed that all signals have very low false detection rates due to a Gaussian noise input.

A frame synchronization technique for IEEE 802.11a and IEEE 802.16 signals that utilized the output of the detection algorithm was demonstrated to be consistent and robust in a multipath fading channel at low SNR values. This allows a simple correction factor to be applied to the frame synchronization result for nearly perfect frame synchronization. Superior frame synchronization will result in lower bit error rates.

An investigation into the packet information available in poor channel environments, where the full packet could not be properly demodulated, suggests that significant packet information is available in the received packet header that could be of interest in the signal intelligence domain.

C. FUTURE WORK

There are several avenues for future work from this thesis. First, work can be done to improve the simulation models. For the IEEE 802.11a model, frequency offset calculation could be added. With the IEEE 802.16 model, a significant amount of future improvements could be made such as channel estimation, frequency offset, and implementing many of the optional features of the standard. Additionally, for both models, functions could be added to emulate layer-two functionality towards building a more comprehensive transmitter/receiver application.

Actual transmitted radio signals could be collected in the lab. The baseband version of the collected signals could be processed by the models. This would allow for a more in-depth investigation into signal detection results for weak signals, as well as a good check of the accuracy of the models.

Finally, further study could be done concerning the statistical characteristics of the decision statistic when the signal is present and when noise is present. This work could be done to tailor a decision statistic for reception in a specific signal intelligence environment.

APPENDIX

This appendix contains the MATLAB code used in the simulations. Figure 39 shows the function call order to run the simulations. The M-file Detector.m is the main file where all parameters are entered and which calls all other functions. This file also processes the results for display. Detector.m calls one of the three signal generation functions depending upon the parameter entered. The output of that function is manipulated by Channels.m depending upon the channel parameter entered. The output of Channels.m is sent to all the detection functions. If any of these functions make a detection, synchronization will be attempted. If synchronization is successful, demodulation will be attempted.

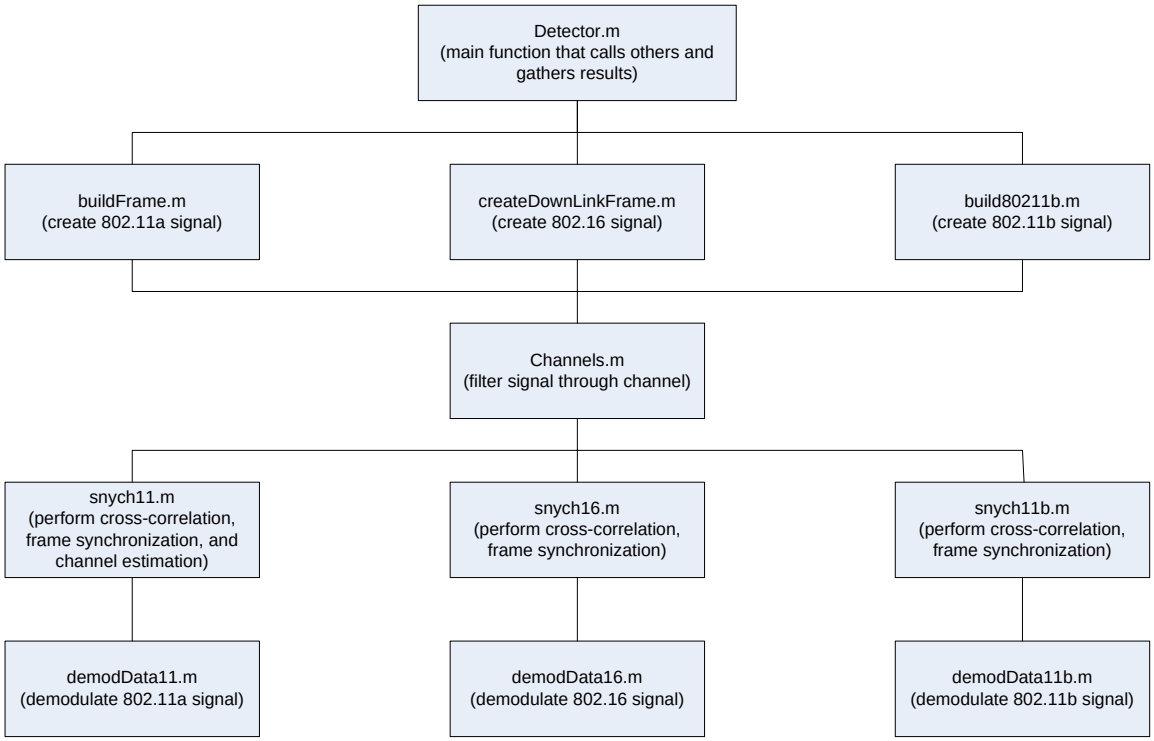


Figure 39. MATLAB Code Function Call Signal Flow for Simulations

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Detector.m
%
%  call functions to generate wireless networking waveform
%  run generated signal through a channel
%  then check for detections and sychronization
%  if so, then demodulate data;
%  Note: does not have complete error checking, so if too few runs or
%  only at low SNRs may cause failures because no signals are be detected
%  and so it has no data to process.
%
%   Input:  (see input parameters below)
%
%   Output:
%       Header BER
%       Data BER
%       Frame synchronization result
%
%   Author:     Keith Howland
%   Created:    23 Jul 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%================================================
% set input parameters
clear;clc;close all
sent_data_length=100;%in bytes;max is 4095 for 802.11a
intro_length=500;
max_run = 3;
max_snr=5;
Th=.3; %detection threshold
interference_variance = .1;
sig='11a'; %802.11a->11a; 802.11b->11b;802.16->16;noise->noise
channel = 'rayleigh'; %'ideal';'awgn';'rayleigh'
sent_data_rate=12; % for 802.16
mod_number_sent=2; % for 802.11
%================================================

%================================================
%set up paths to directories for different signals
addpath('H:\thesis\Detector\test\80211\Transmitter')
```

```
addpath('H:\thesis\Detector\test\80216\Transmitter')
addpath('H:\thesis\Detector\test\11b\Transmitter')
addpath('H:\thesis\Detector\test\80211\Receiver')
addpath('H:\thesis\Detector\test\80216\Receiver')
addpath('H:\thesis\Detector\test\11b\Receiver');
%================================================

%================================================
%run simulation so max_run times for each SNR up to max_snr
for SNR = 0:max_snr
   for run = 1:max_run
      %tic
      disp([num2str(SNR) ' ' num2str(run)])   %where am I
         %create input signal
      if strcmp(sig,'11a')
         [signal,PLCP_bits,data_bits] = ...
            buildFrame(sent_data_rate,sent_data_length);
         preamble_length = 320;
      elseif strcmp(sig,'16')
         [signal,fch_bits,data_bits] = ...
            createDownLinkFrame( mod_number_sent, sent_data_length);
         preamble_length = 640;
      elseif strcmp(sig,'11b')
         [signal,sent_plcp,data_bits] =  build80211b(sent_data_length);
         signal = signal.'*.1;%match power to other signals
         preamble_length = 144;
      else strcmp(sig,'noise');
         signal = complex(randn(sent_data_length,1),...
            randn(sent_data_length,1))*.01;
      end

      %add interference beginning
      [output, actual_start_of_data] = beginningInterference(signal, ...
         preamble_length,intro_length,interference_variance);

      %go through channel
      chan_output = Channels(output,channel,SNR);

      % go to receivers
      %11a
      [signal_detected11a(run,SNR+1),start_of_data11a,H] = ...
         synch11(chan_output,Th);

      if signal_detected11a(run,SNR+1) == 1
         [rec_data_80211,rec_signal,parity_bit(run,SNR+1), ...
```

```
        rec_data_rate(run,SNR+1), rec_data_length(run,SNR+1),...
        rec_tail(run,SNR+1)] = demodData11(chan_output,...
            start_of_data11a,H,sent_data_rate, sent_data_length);

    frame_offset11(run,SNR+1) = start_of_data11a - ...
        actual_start_of_data;

    if strcmp(sig,'11a') ~=1
        continue
    elseif rec_data_80211 == 0
        rec_data_80211 = zeros(length(data_bits),1);
    end

    [er,BER_PLCP11a(run,SNR+1)] = biterr(PLCP_bits,rec_signal');
    [num_error, BER_80211data(run,SNR+1)] = ...
        biterr(data_bits,rec_data_80211);
end

%16
[signal_detected16(run,SNR+1),H,start_of_data16] = ...
    synch16(chan_output,Th);
frame_offset16(run,SNR+1) = start_of_data16 - actual_start_of_data;
if signal_detected16(run,SNR+1) ==1
    [rec_data_80216,rec_fch,HCS_check(run,SNR+1)] = ...
        demodData16(chan_output,H, ...
        start_of_data16, mod_number_sent);
    if strcmp(sig,'16') ~=1
        continue
    elseif rec_data_80216 == 0
        rec_data_80216 = zeros(length(data_bits),1);
    end
    [fch_be, BER_fch(run,SNR+1)] = biterr(fch_bits,rec_fch');
    [data_be, BER_80216data(run,SNR+1)] = ...
        biterr(data_bits(:),rec_data_80216(:));
end

%11b
[signal_detected11b(run,SNR+1),start_of_data11b] = ...
    synch11b(chan_output,Th);
frame_offset11b(run,SNR+1) = ...
    start_of_data11b - actual_start_of_data;
if signal_detected11b(run,SNR+1) ==1
    [data_rate, num_data_octets,rec_data,rec_plcp,...
        CRC_result11b(run,SNR+1)] = ...
        demodData11b(chan_output,start_of_data11b);
```

```
        if strcmp(sig,'11b') ~=1
            continue
        elseif rec_data == 0
            rec_data = zeros(length(data_bits),1);
        elseif length(data_bits) ~= length(rec_data)
            pad = length(data_bits) - length(rec_data);
            rec_data = [rec_data; zeros(pad,1)];
        end
        %calculate BER
        [plcp_11b_errors,BER_PLCP11b(run,SNR+1)] = ...
            biterr(sent_plcp,rec_plcp);
        [data11b_errors,BER_80211bdata(run,SNR+1)] = ...
            biterr(data_bits,rec_data');
    end
%toc
  end    %end run loop
end    %end SNR Loop
%=================================================================
==========

%=================================================================
==========
%RESULTS
%=================================================================
==========
%this is Pd when signal is present
%and Pfa when noise is present
prob_detect_11a = sum(signal_detected11a)/max_run;
prob_detect_16 = sum(signal_detected16)/max_run;
prob_detect_11b = sum(signal_detected11b)/max_run;
figure;plot(0:max_snr,prob_detect_11a,'*-',0:max_snr, ...
  prob_detect_16,'d-', 0:max_snr,prob_detect_11b,'s-');
legend('802.11a/g','802.16','802.11b');
xlabel('SNR (dB)');
if strcmp(sig,'noise')
  ylabel('Probability of False Alarm')
else
  ylabel('Probability of Detection')
end

%only run these results if signal was present
if strcmp(sig,'noise')
  return
elseif strcmp(sig,'11a')
```

```
    prob_miss_11a = (max_run - sum(signal_detected11a))/max_run;
    prob_parity_okay = sum(parity_bit)/max_run;

    figure;plot(0:max_snr,prob_miss_11a,'*-');legend('802.11a')
    xlabel('SNR (dB)');ylabel('Probability of Miss')

    figure;plot(0:max_snr,prob_detect_11a,'*-',0:max_snr,...
       prob_parity_okay,'d-');
    legend('Signal Detected','Parity Check Good');
    xlabel('SNR (dB)');ylabel('Probability of Occurence')

    sd11a = signal_detected11a ==1;%make logicals
     %only take those BERs where signal was detected
    for i = 1:size(sd11a,2)
       c = BER_PLCP11a(:,i);
       d = BER_80211data(:,i);
       fo = frame_offset11(:,i);
       fo2=fo(sd11a(:,i));
       e = c(sd11a(:,i));
       f= d(sd11a(:,i));
       avg_offset(i) = mean(fo2);
       avg_BER_PLCP(i) = mean(e);
       avg_BER_80211(i) = mean(f);
    end

    %make table for 80211a/g results:  SNR,Parity,BER_PLCP,BER_DATA
    result11(:,1)=0:max_snr;
    result11(:,2)=prob_detect_11a;
    result11(:,3)=prob_parity_okay';
    result11(:,4)= avg_offset;
    result11(:,5)=avg_BER_PLCP;
    result11(:,6)=avg_BER_80211;

    %make table of SIGNAL content for specific SNR
    rec_SIG(:,1) = rec_data_rate(:,1);
    rec_SIG(:,2) = rec_data_length(:,1);
    rec_SIG(:,3) = rec_tail(:,1);

    result11
    rec_SIG
elseif strcmp(sig,'16')
    prob_miss_16 = (max_run - sum(signal_detected16))/max_run;
    prob_HCS = sum(HCS_check)/max_run;

   figure;plot(0:max_snr,prob_detect_16,'d-',0:max_snr, ...
```

```
    prob_detect_11a,'*-', 0:max_snr,prob_detect_11b,'s-', ...
    0:max_snr,prob_HCS,'o-');
    legend('802.16 Detection','802.11a Detection','802.11b Detection', ...
    'Good CRC');
    xlabel('SNR (dB)');ylabel('Probability of Occurence')

    sd16 = signal_detected16 ==1;
    for i = 1:size(sd16,2)
       fo = frame_offset16(:,i);
       fo2=fo(sd16(:,i));
       avg_offset16(i) = mean(fo2);
    end
    avg_offset16

elseif strcmp(sig,'11b')
    prob_miss_11b = (max_run - sum(signal_detected11b))/max_run;
    prob_CRC11b = sum(CRC_result11b)/max_run;

    sd11b = signal_detected11b ==1;%make logicals
     %only take those BERs where signal was detected
    for i = 1:size(sd11b,2)
       c = BER_PLCP11b(:,i);
       d = BER_80211bdata(:,i);
       fo = frame_offset11b(:,i);
       fo2=fo(sd11b(:,i));
       e = c(sd11b(:,i));
       f= d(sd11b(:,i));
       avg_offset11b(i) = mean(fo2);
       avg_BER_PLCP11b(i) = mean(e);
       avg_BER_80211b(i) = mean(f);
    end

    %make table for 80211a/g results:  SNR,Parity,BER_PLCP,BER_DATA
    result11b(:,1)=0:max_snr;
    result11b(:,2)=prob_detect_11b;
    result11b(:,3)=prob_CRC11b;
    result11b(:,4)= avg_offset11b;
    result11b(:,5)=avg_BER_PLCP11b;
    result11b(:,6)=avg_BER_80211b;
    result11b

    figure;plot(0:max_snr,prob_detect_11b,'d-',0:max_snr, ...
    prob_detect_11a,'*-', 0:max_snr,prob_detect_16,'s-', ...
    0:max_snr,prob_CRC11b,'o-');
    legend('802.11b Detection','802.11a Detection','802.16 Detection', ...
```

```
    'Good CRC');
    xlabel('SNR (dB)');ylabel('Probability of Occurence')
end
```

```
function [output] =  Channels(input,channel,SNR)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% Channels.m
%
%  run OFDM time domain samples through channels
%
%   Input:                                                              %
%       input:      OFDM time domain samples
%       channel:    either awgn or Rayleigh as text string
%       SNR:        signal to noise ratio
%   Output:                                                             %
%       output:     channel filtered time domain samples
%
%   Author:     Keith Howland
%   Created:    03 May 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%==================================%
%send signal through channel
%==================================%
if strcmp('awgn',channel)
   %Additive White Gaussian Noise Channel
   output = awgn(input,SNR,'measured');
elseif strcmp('rayleigh',channel)
   %Rayleigh Channel with Additive White Gaussian Noise
   chan = rayleighchan(1/(20*10^6),50);
   chan.pathdelays=[0 1e-7];
   chan.avgpathgaindb=[0 0];
   y = filter(chan,input);
   output=awgn(y,SNR,'measured');
else
   output = input;
end
```

```
function [output,PLCP_bits,data] = ...
   buildFrame(sent_data_rate, sent_data_length)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% buildFrame.m
%
%  Create OFDM time domain signal samples for transmission
%
%   Input:                                                              %
%      sent_data_rate:     sent data rate
%      sent_data_length:   length in bytes of data from MAC layer
%   Output:                                                             %
%      output:        OFDM time domain signal samples
%      PLCP_bits:      PLCP signal field bits
%      data:          original data bits from MAC layer
%
%   Author:     Keith Howland
%   Created:    29 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%=================================%
%data bits are random ones and zeros
%build PLCP Header
%make time domain OFDM samples
%=================================%
data = randint(sent_data_length*8,1);
[PLCP,PLCP_bits] = buildPLCPHeader(sent_data_length,sent_data_rate);
time_sig = makeData(data,sent_data_length,sent_data_rate);

%get preambles
load short_preamble_time_domain
load long_preamble_time_domain

%=================================%
%final time frame starts with short and long preambles then signal field
%then data
%=================================%
output = [short_preamble_time_domain(1:160); ...
```

```
short_preamble_time_domain(161)+long_preamble_time_domain(1);...
long_preamble_time_domain(2:160); long_preamble_time_domain(161) ...
+ PLCP(1); PLCP(2:80); PLCP(81) + time_sig(1); time_sig(2:end)];
```

```
function [PLCP_Header,PLCP_bits] = buildPLCPHeader(length,data_rate)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% buildPLCPHeader.m
%
%  this function constructs the PLCP Header per the 802.11a standard;
%  called by buildFrame
%
%   Input:                                                              %
%      length:           length in octets of MAC layer input
%      data_rate:         transmission data rate
%
%   Output:                                                             %
%      PLCP_Header:       full OFDM symbol of PLCP header
%      PLCP_bits:         bits of the PLCP signal field for comparison
%                         with received signal field bits
%
%   Author:     Keith Howland
%   Created:    20 Feb 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%this section constructs the contents of the SIGNAL field of 802.11a
%standard

if data_rate == 6
   signal(1:4) = [1 1 0 1];
elseif data_rate == 9
   signal(1:4) = [1 1 1 1];
elseif data_rate == 12
   signal(1:4) = [0 1 0 1];
elseif data_rate == 18
   signal(1:4) = [0 1 1 1];
elseif data_rate == 24
   signal(1:4) = [1 0 0 1];
elseif data_rate == 36
   signal(1:4) = [1 0 1 1];
elseif data_rate == 48
   signal(1:4) = [0 0 0 1];
elseif data_rate == 54
   signal(1:4) = [0 0 1 1];
end
```

```
signal(5) =0; %reserved in standard

PLCP_length = fliplr((dec2bin(length,12)));    %LSB bit first per standard
for i = 1:12;                                  %field length is 12
      signal(5+i) = str2double(PLCP_length(i));     %convert str to num;
end


%parity calculation
if rem(sum(signal(1:17)),2)== 0
   signal(18) = 0;
else
   signal(18) =1;
end

signal(19:24) = zeros(1,6);  %Signal tail bits; all bits to zero
PLCP_bits = signal;
%always encode at 1/2 rate; encoder state is 0
coded_signal = ConvolutionCoding11(signal,.5);

%next data is interleaved
interleaved = interleaver11(coded_signal,48,1); %1 is Number of data bits
                                    % per subcarrier
modulated_signal = modData11(interleaved,2); %always modulate with BPSK

%create symbol
symbol = createSymbol11(modulated_signal,1);
symbol=symbol';
%get time domain
time_symbol = ifft(symbol);
%cyclically extend and window to match Annex G example
PLCP_Header = [time_symbol(49:64); time_symbol ;time_symbol(1)];
PLCP_Header(1) = PLCP_Header(1)/2;%window to match Appendix G of standard
PLCP_Header(81) = PLCP_Header(81)/2;
```

```
function output = ConvolutionCoding11(bits,code_rate)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% ConvolutionCoding11.m
%
%  performs convolution encoding for rates 1/2;
%  for 2/3 and 3/4 by puncturing the 1/2 coded data;
%  creates encoder for constraint length 7 and
%  generator polynomials 133 and 177 as per standard p.16 of PHY
%
%   Input:                                                              %
%      bits:          scrambled bits
%      code_rate:      rate for coding:1/2,3/4,2/3
%
%   Output:                                                             %
%      output:        encoded bits
%
%   Author:     Keith Howland
%   Created:    20 Apr 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


trellis = poly2trellis(7,[133 171]);

if code_rate == .5  %rate 1/2 encoding
   output = convenc(bits,trellis);
elseif code_rate == .66 %rate 2/3 through puncturing 1/2
   output = convenc(bits,trellis,[1 1 1 0]);
elseif code_rate == .75  %rate 3/4 through puncturing 1/2
   output = convenc(bits,trellis,[1 1 1 0 0 1]);
end
```

```
function [OFDM_symbol] = createSymbol11(modulated_data,pilot_index)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% createSymbol.m
%
%  create the OFDM symbol; put data into the symbol.  Then add pilots and
%  nulls.  then map values for IFFT per standard
%
%   Input:                                                              %
%      modulated_data:     symbol mapped complex values
%      pilot_index:        index where pilot is being placed in data
%
%   Output:                                                             %
%      OFDM_symbol:        OFDM symbol
%
%   Author:     Keith Howland
%   Created:    15 Feb 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

symbol(1,:) = [-32:-1 0 1:31];    %this is just for my visualization
symbol(2,:) = zeros(1,64); %reserve space

%add data
symbol(2,[7:11,13:25,27:32,34:39,41:53,55:59])= modulated_data;

%add pilots
symbol  = insertPilot(symbol(2,:),pilot_index);

%remap for IFFT per figure 109 in standard
OFDM_symbol(1)=symbol(33);
OFDM_symbol(2:27)=symbol(34:59);
OFDM_symbol(28:38)=zeros(1,11);
OFDM_symbol(39:64)=symbol(7:32);
```

```
function [data] = insertPilot(data,i)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% insertPilot.m
%
%  this function inserts pilot symbols.  Four pilots per symbol.
%  cyclic through the 32 sets
%  Note:%the pilots for the signal field are hard coded in, so i starts
%  with 2
%
%   Input:                                                              %
%      data:          symbol mapped values
%      i:             index where pilot is being placed in data
%
%   Output:                                                             %
%      data:        OFDM symbol containing pilots in the correct indices
%
%   Author:     Keith Howland
%   Created:    22 Feb 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


pn = [1,1,1,1,-1,-1,-1,1,-1,-1,-1,-1,1,1,-1,1,-1,-1,1,1,-1,1,1,-1,1,1,...
    1,1,1,1,-1,1,1,1,-1,1,1,-1,-1,1,1,1,-1,1,-1,-1,-1,1,-1,1,-1,-1,1,...
    -1,-1,1,1,1,1,1,1,-1,-1,1,1,-1,-1,1,-1,1,-1,1,1,1,-1,-1,-1,1,1,-1,-1,...
    -1,-1,1,-1,-1,1,-1,1,1,1,1,-1,1,-1,1,-1,1,-1,-1,-1,-1,-1,1,-1,1,1,...
    -1,1,-1,1,1,1,-1,-1,1,-1,-1,-1,1,1,1,-1,-1,-1,-1,-1,-1,-1];

%cycle through pn codes
if i > 127
   i = rem(i,127);
   if i ==0;
      i = 127;
   end
end

%insert pilots into data
data(12)=pn(i)*1;
data(26)=pn(i)*1;
data(40)=pn(i)*1;
data(54)=pn(i)*-1;
```

```
function [output] = interleaver11(bits,Ncbps,Nbpsc)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% interleaver11.m
%
%  %this function performs data interleaving per page 17 of PHY section of
%  802.11a standard
%  these were calculated using interleaverCalculator.m
%
%   Input:                                                              %
%      bits:          encoded bits
%      Ncbps:         number of coded bits per symbol
%      Nbpsc:         number of coded bits per subcarrier
%
%   Output:                                                             %
%      output:        interleaved bits
%
%   Author:     Keith Howland
%   Created:    15 Feb 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


if Ncbps == 48 && Nbpsc ==1
   first =bits([1,17,33,2,18,34,3,19,35,4,20,36,5,21,37,6,22,38,7,23, ...
      39,8,24,40,9,25,41,10,26,42,11,27,43,12,28,44,13,29,45,14,30, ...
      46,15,31,47,16,32,48]);
   output = first([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19, ...
      20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39, ...
      40,41,42,43,44,45,46,47,48]);
elseif Ncbps == 96 && Nbpsc ==2
   first = bits([1,17,33,49,65,81,2,18,34,50,66,82,3,19,35,51,67,83, ...
      4,20,36,52,68,84,5,21,37,53,69,85,6,22,38,54,70,86,7,23,39,55, ...
      71,87,8,24,40,56,72,88,9,25,41,57,73,89,10,26,42,58,74,90,11, ...
      27,43,59,75,91,12,28,44,60,76,92,13,29,45,61,77,93,14,30,46,62, ...
      78,94,15,31,47,63,79,95,16,32,48,64,80,96]);
   output = first([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19, ...
      20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39, ...
      40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59, ...
      60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79, ...
      80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96]);
elseif Ncbps == 192 && Nbpsc == 4
   first = bits([1,17,33,49,65,81,97,113,129,145,161,177,2,18,34,50, ...
```

```
        66,82,98,114,130,146,162,178,3,19,35,51,67,83,99,115,131,147, ...
        163,179,4,20,36,52,68,84,100,116,132,148,164,180,5,21,37,53, ...
        69,85,101,117,133,149,165,181,6,22,38,54,70,86,102,118,134, ...
        150,166,182,7,23,39,55,71,87,103,119,135,151,167,183,8,24,40, ...
        56,72,88,104,120,136,152,168,184,9,25,41,57,73,89,105,121,137, ...
        153,169,185,10,26,42,58,74,90,106,122,138,154,170,186,11,27, ...
        43,59,75,91,107,123,139,155,171,187,12,28,44,60,76,92,108, ...
        124,140,156,172,188,13,29,45,61,77,93,109,125,141,157,173, ...
        189,14,30,46,62,78,94,110,126,142,158,174,190,15,31,47,63,79, ...
        95,111,127,143,159,175,191,16,32,48,64,80,96,112,128,144,160, ...
        176,192]);
    output= first([1,2,3,4,5,6,7,8,9,10,11,12,14,13,16,15,18,17,20,19, ...
        22,21,24,23,25,26,27,28,29,30,31,32,33,34,35,36,38,37,40,39, ...
        42,41,44,43,46,45,48,47,49,50,51,52,53,54,55,56,57,58,59,60, ...
        62,61,64,63,66,65,68,67,70,69,72,71,73,74,75,76,77,78,79,80, ...
        81,82,83,84,86,85,88,87,90,89,92,91,94,93,96,95,97,98,99,100, ...
        101,102,103,104,105,106,107,108,110,109,112,111,114,113,116, ...
        115,118,117,120,119,121,122,123,124,125,126,127,128,129,130, ...
        131,132,134,133,136,135,138,137,140,139,142,141,144,143,145, ...
        146,147,148,149,150,151,152,153,154,155,156,158,157,160,159, ...
        162,161,164,163,166,165,168,167,169,170,171,172,173,174,175, ...
        176,177,178,179,180,182,181,184,183,186,185,188,187,190,189, ...
        192,191]);
elseif Ncbps == 288 && Nbpsc == 6
    first = bits([1,17,33,49,65,81,97,113,129,145,161,177,193,209,225, ...
        241,257,273,2,18,34,50,66,82,98,114,130,146,162,178,194,210, ...
        226,242,258,274,3,19,35,51,67,83,99,115,131,147,163,179,195, ...
        211,227,243,259,275,4,20,36,52,68,84,100,116,132,148,164,180, ...
        196,212,228,244,260,276,5,21,37,53,69,85,101,117,133,149,165, ...
        181,197,213,229,245,261,277,6,22,38,54,70,86,102,118,134,150, ...
        166,182,198,214,230,246,262,278,7,23,39,55,71,87,103,119,135, ...
        151,167,183,199,215,231,247,263,279,8,24,40,56,72,88,104,120, ...
        136,152,168,184,200,216,232,248,264,280,9,25,41,57,73,89,105, ...
        121,137,153,169,185,201,217,233,249,265,281,10,26,42,58,74,90, ...
        106,122,138,154,170,186,202,218,234,250,266,282,11,27,43,59, ...
        75,91,107,123,139,155,171,187,203,219,235,251,267,283,12,28, ...
        44,60,76,92,108,124,140,156,172,188,204,220,236,252,268,284,13, ...
        29,45,61,77,93,109,125,141,157,173,189,205,221,237,253,269,285, ...
        14,30,46,62,78,94,110,126,142,158,174,190,206,222,238,254,270, ...
        286,15,31,47,63,79,95,111,127,143,159,175,191,207,223,239,255, ...
        271,287,16,32,48,64,80,96,112,128,144,160,176,192,208,224,240, ...
        256,272,288]);
    output = first([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,20,21, ...
        19,23,24,22,26,27,25,29,30,28,32,33,31,35,36,34,39,37,38,42,40, ...
        41,45,43,44,48,46,47,51,49,50,54,52,53,55,56,57,58,59,60,61,62, ...
```

```
    63,64,65,66,67,68,69,70,71,72,74,75,73,77,78,76,80,81,79,83,84, ...
    82,86,87,85,89,90,88,93,91,92,96,94,95,99,97,98,102,100,101, ...
    105,103,104,108,106,107,109,110,111,112,113,114,115,116,117, ...
    118,119,120,121,122,123,124,125,126,128,129,127,131,132,130, ...
    134,135,133,137,138,136,140,141,139,143,144,142,147,145,146, ...
    150,148,149,153,151,152,156,154,155,159,157,158,162,160,161, ...
    163,164,165,166,167,168,169,170,171,172,173,174,175,176,177, ...
    178,179,180,182,183,181,185,186,184,188,189,187,191,192,190, ...
    194,195,193,197,198,196,201,199,200,204,202,203,207,205,206, ...
    210,208,209,213,211,212,216,214,215,217,218,219,220,221,222, ...
    223,224,225,226,227,228,229,230,231,232,233,234,236,237,235, ...
    239,240,238,242,243,241,245,246,244,248,249,247,251,252,250, ...
    255,253,254,258,256,257,261,259,260,264,262,263,267,265,266, ...
    270,268,269,271,272,273,274,275,276,277,278,279,280,281,282, ...
    283,284,285,286,287,288]);
end
```

```
function [time_matrix] = makeData(input,len,data_rate)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% makeData.m
%
%  this function works on the data bits not including the Header.  From
%  data rate you get all necessary information for coding.
%  Nbpsc:   %Number of coded bits per OFDM symbol
%  Ncbps:   %Number of data bits per OFDM symbol
%  modulation: modulation scheme described by M-ary number:
%                2=BPSK; 4=QPSK; 16=16_QAM 64=64_QAM
%  coding_rate:  %convolution encoding rate
%  Then service bits and tail bits are added to complete input stream
%  then data is padded to match IFFT size.  Then data is scrambled, coded,
%  interleaved,modulated.  Pilots and nulls are inserted.  Then IFFT is
%  taken and cyclic prefix is prepended.
%
%   Input:                                                              %
%      input:            data bits from MAC layer
%      len:              number of data bits from MAC layer
%      data_rate:        transmission data rate
%
%   Output:                                                             %
%      time_matrix:       time domain sammple values including cyclic
%                         prefix
%
%   Author:     Keith Howland
%   Created:    15 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


if data_rate == 6
   Nbpsc = 1;
   Ncbps =48;
   Ndbps =24;
   modulation = 2;
   coding_rate = .5;
 elseif data_rate ==9;
    Nbpsc = 1;
    Ncbps =48;
   Ndbps =36;
   modulation = 2;
```

```
    coding_rate = .75;
elseif data_rate==12;
    Nbpsc = 2;
    Ncbps =96;
    Ndbps =48;
    modulation = 4;
    coding_rate = .5;
 elseif data_rate ==18;
    Nbpsc = 2;
    Ncbps =96;
    Ndbps =72;
    modulation = 4;
    coding_rate = .75;
 elseif data_rate==24;
    Nbpsc = 4;
    Ncbps =192;
    Ndbps =96;
    modulation = 16;
    coding_rate = .5;
 elseif data_rate==36;
    Nbpsc = 4;
    Ncbps =192;
    Ndbps =144;
    modulation = 16;
    coding_rate = .75;
 elseif data_rate==48;
    Nbpsc = 6;
    Ncbps =288;
    Ndbps =192;
    modulation = 64;
    coding_rate = .66;
 elseif data_rate==54;
    Nbpsc =6;
    Ncbps =288;
    Ndbps =216;
    modulation = 64;
    coding_rate = .75;
end

%reshape data so block length after encoding and modulation is 48 bits
%pad the data so coded data matches FFT size; see section 17.3.5.4

Nsym = ceil((16+8*len+6)/Ndbps);
Ndata = Nsym * Ndbps;
Npad = Ndata - (16+8*len+6);
```

```
%prepend SERVICE bits (16 zeros)
%and postpend PPDU tail bits (6 zeros) to data
DATA = [zeros(16,1); input; zeros(6,1); zeros(Npad,1) ];

scrambled_data = Scrambler(DATA);
%tail bits are replaced with non-scrambled zero bits; section 17.3.5.2
scrambled_data(length(input)+16+1:length(input)+16+1+6-1) = zeros(1,6);
coded_data = ConvolutionCoding11(scrambled_data,coding_rate);

DATA_blocks = reshape(coded_data,Ncbps,Nsym); %reshape into blocks

%take each block of data and interleave it, modulated it, insert
%piolts; freq_matrix is final data in frequency form
time_matrix=zeros(80,Nsym);
for i =1:Nsym
   block = DATA_blocks(:,i);   %take first block
   int_block = interleaver11(block,Ncbps,Nbpsc); %interleave
   mod_block = modData11(int_block,modulation);      %modulate
   %because signal pilot hard coded in PLCP so start at 2: (i+1)
   symbol = createSymbol11(mod_block,i+1);
   time_symbol = ifft(symbol,64);       %bring into time domain
   time_matrix(:,i) = [time_symbol(49:64) time_symbol];%add cyclic prefix
end
time_matrix=time_matrix(:);
```

```
function [output] = modData11(bits,mod)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% modData11.m
%
%  this function takes the input bits and modulates them according to given
%  modulation technique: BPSK, QPSK, 16-QAM, 64-QAM
%  Kmod is a normalizatio factor to achieve the same average power for all
%  mappings
%  Note: this only works with MATLAB version R2007a or higher
%  Note:replace sqrt with actual numbers for faster performance
%
%   Input:                                                              %
%      bits:          interleaved bits
%      mod:            modulation schemes by their M-ary number;
%
%   Output:                                                             %
%      output:        complex valued symbols
%
%   Author:     Keith Howland
%   Created:    15 Feb 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%BPSK
if mod == 2
   a = bits ==0;
   bits(a)=-1;
   %for BPSK Kmod = 1;
   output = bits;
end

%QPSK
if mod == 4
   bits = reshape(bits,2,length(bits)/2);
   g = modem.qammod('M', 4, 'PhaseOffset', 0, 'SymbolOrder',...
   'user-defined','SymbolMapping',[1 0 3 2], 'InputType', 'bit');
   qpsk = modulate(g,bits);
   Kmod = 1/sqrt(2);
   output = qpsk*Kmod;
end
```

```
%16_QAM
if mod == 16
   bits = reshape(bits,4,length(bits)/4);

   g = modem.qammod('M',16, 'SymbolOrder','user-defined', ...
      'SymbolMapping',[2 3 1 0 6 7 5 4 14 15 13 12 10 11 9 8], ...
      'InputType','bit');  % Create a modulator object
   qam16 = modulate(g,bits);   % modulate the signal y.
   Kmod = 1/sqrt(10);
   output = qam16*Kmod;
end

%64-QAM
if mod == 64
   bits = reshape(bits,6,length(bits)/6);
   g = modem.qammod('M',64,'SymbolOrder','user-defined', ...
      'SymbolMapping',[4 5 7 6 2 3 1 0 12 13 15 14 10 11 9 8 28 29 ...
      31 30 26 27 25 24 20 21 23 22 18 19 17 16 52 53 55 54 50 51 49 ...
      48 60 61 63 62 58 59 57 56 44 45 47 46 42 43 41 40 36 37 39 38 ...
      34 35 33 32], 'InputType','bit');  % Create a modulator object
   qam64 = modulate(g,bits);   % modulate the signal y.
   Kmod = 1/sqrt(42);
   output = qam64*Kmod;
end
```

```
function [output] = Scrambler(input)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% Scrambler.m
%
%  this function scrambles the data bits as per page 16 of 802.11a
%  generator polynomial: S(x) = x^7 + x^4 + 1
%  note: need to determine how to set initial state
%
%   Input:                                                              %
%      input:         data bits from MAC layer
%
%   Output:                                                             %
%      output:        scrambled bits
%
%   Author:     Keith Howland
%   Created:    20 Apr 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


output=zeros(1,length(input));
%set initial state
reg = [1 0 1 1 1 0 1];
for i = 1:length(input)
   new = xor(reg(4),reg(7));
   %shift bits left one
   reg = [new reg([1 2 3 4 5 6])];
   % modulo 2 add output with data bit
   output(i) = xor(new,input(i));
end

%for testing
%load first144testvector
```

```
function [data_bits] = deCode11(bits,code_rate)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% deCode11.m
%
%   Deconvolutional code the received bits given code rate read from
%   SIGNAL field
%
%   Input:                                                              %
%       bits:       coded OFDM symbol bits
%       code_rate:  code rate per SIGNAL field for symbol
%   Output:                                                             %
%       data_bits:  data bits for symbol
%
%   Author:     Keith Howland
%   Created:    23 Feb 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%=======================================%
%set up trellis for decoding
%constraint length 7
%generator polynomicals: 133 & 171
%table length not specified in standard
%=======================================%
trellis = poly2trellis(7,[133 171]);
table_length = 3;

%=======================================%
%use viterbi decoding based on code rate
% 2/3 and 3/4 need to be depunctured
%could also implement soft decoding
%=======================================%
if code_rate == .5
   data_bits  = vitdec(bits,trellis,table_length, 'trunc','hard');
elseif code_rate == .66
   data_bits  = vitdec(bits,trellis,table_length, 'trunc','hard',[1 1 1 0]);
elseif code_rate == .75
   data_bits  = vitdec(bits,trellis,table_length, 'trunc','hard',[1 1 1 0 0 1]);
end
```

```
%for testing
%[n1,r1] = biterr(z(3+1:end),x(1:end-3));
```

```
function [output] = deinterleaver11(bits,Ncbps,Nbpsc)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% deinterleaver11.m
%
%   this function performs data deinterleaving per page 17 of PHY section
%   of 802.11a standard.  Indicies were calculated using
%   interleaverCalculator.m
%
%   Input:                                                              %
%       bits: vector of bits to be interleaved
%       Nbpsc: coded bits per subcarrier
%       Ncbps: coded bits per OFDM symbol
%   Output:                                                             %
%       data_bits: de-interleaved data bits for symbol
%
%   Author:     Keith Howland
%   Created:    03 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


if Ncbps == 48 && Nbpsc ==1
    first =bits([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21, ...
        22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42, ...
        43,44,45,46,47,48]);
    output = first([1,4,7,10,13,16,19,22,25,28,31,34,37,40,43,46,2,5,8, ...
        11,14,17,20,23,26,29,32,35,38,41,44,47,3,6,9,12,15,18,21,24,27, ...
        30,33,36,39,42,45,48]);
elseif Ncbps == 96 && Nbpsc ==2
    first = bits([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...
        22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42, ...
        43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63, ...
        64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84, ...
        85,86,87,88,89,90,91,92,93,94,95,96]);
    output = first([1,7,13,19,25,31,37,43,49,55,61,67,73,79,85,91,2,8,...
        14,20,26,32,38,44,50,56,62,68,74,80,86,92,3,9,15,21,27,33,39,...
        45,51,57,63,69,75,81,87,93,4,10,16,22,28,34,40,46,52,58,64,70,...
        76,82,88,94,5,11,17,23,29,35,41,47,53,59,65,71,77,83,89,95,6,...
        12,18,24,30,36,42,48,54,60,66,72,78,84,90,96]);
elseif Ncbps == 192 && Nbpsc == 4
    first = bits([1,2,3,4,5,6,7,8,9,10,11,12,14,13,16,15,18,17,20,19,22,...
```

```
        21,24,23,25,26,27,28,29,30,31,32,33,34,35,36,38,37,40,39,42,41,...
        44,43,46,45,48,47,49,50,51,52,53,54,55,56,57,58,59,60,62,61,64,...
        63,66,65,68,67,70,69,72,71,73,74,75,76,77,78,79,80,81,82,83,84,...
        86,85,88,87,90,89,92,91,94,93,96,95,97,98,99,100,101,102,103,...
        104,105,106,107,108,110,109,112,111,114,113,116,115,118,117,120,...
        119,121,122,123,124,125,126,127,128,129,130,131,132,134,133,136,...
        135,138,137,140,139,142,141,144,143,145,146,147,148,149,150,151,...
        152,153,154,155,156,158,157,160,159,162,161,164,163,166,165,168,...
        167,169,170,171,172,173,174,175,176,177,178,179,180,182,181,184,...
        183,186,185,188,187,190,189,192,191]);
    output= first([1,13,25,37,49,61,73,85,97,109,121,133,145,157,169,...
        181,2,14,26,38,50,62,74,86,98,110,122,134,146,158,170,182,3,15,...
        27,39,51,63,75,87,99,111,123,135,147,159,171,183,4,16,28,40,52,...
        64,76,88,100,112,124,136,148,160,172,184,5,17,29,41,53,65,77,89,...
        101,113,125,137,149,161,173,185,6,18,30,42,54,66,78,90,102,114,...
        126,138,150,162,174,186,7,19,31,43,55,67,79,91,103,115,127,139,...
        151,163,175,187,8,20,32,44,56,68,80,92,104,116,128,140,152,164,...
        176,188,9,21,33,45,57,69,81,93,105,117,129,141,153,165,177,189,...
        10,22,34,46,58,70,82,94,106,118,130,142,154,166,178,190,11,23,...
        35,47,59,71,83,95,107,119,131,143,155,167,179,191,12,24,36,48,...
        60,72,84,96,108,120,132,144,156,168,180,192]);
elseif Ncbps == 288 && Nbpsc == 6
    first = bits([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,21,19,20,...
        24,22,23,27,25,26,30,28,29,33,31,32,36,34,35,38,39,37,41,42,40,...
        44,45,43,47,48,46,50,51,49,53,54,52,55,56,57,58,59,60,61,62,63,...
        64,65,66,67,68,69,70,71,72,75,73,74,78,76,77,81,79,80,84,82,83,...
        87,85,86,90,88,89,92,93,91,95,96,94,98,99,97,101,102,100,104,...
        105,103,107,108,106,109,110,111,112,113,114,115,116,117,118,...
        119,120,121,122,123,124,125,126,129,127,128,132,130,131,135,...
        133,134,138,136,137,141,139,140,144,142,143,146,147,145,149,...
        150,148,152,153,151,155,156,154,158,159,157,161,162,160,163,...
        164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,...
        179,180,183,181,182,186,184,185,189,187,188,192,190,191,195,...
        193,194,198,196,197,200,201,199,203,204,202,206,207,205,209,...
        210,208,212,213,211,215,216,214,217,218,219,220,221,222,223,...
        224,225,226,227,228,229,230,231,232,233,234,237,235,236,240,...
        238,239,243,241,242,246,244,245,249,247,248,252,250,251,254,...
        255,253,257,258,256,260,261,259,263,264,262,266,267,265,269,...
        270,268,271,272,273,274,275,276,277,278,279,280,281,282,283,...
        284,285,286,287,288]);
    output = first([1,19,37,55,73,91,109,127,145,163,181,199,217,235,...
        253,271,2,20,38,56,74,92,110,128,146,164,182,200,218,236,254,...
        272,3,21,39,57,75,93,111,129,147,165,183,201,219,237,255,273,4,...
        22,40,58,76,94,112,130,148,166,184,202,220,238,256,274,5,23,41,...
        59,77,95,113,131,149,167,185,203,221,239,257,275,6,24,42,60,78,...
```

```
    96,114,132,150,168,186,204,222,240,258,276,7,25,43,61,79,97,115,...
    133,151,169,187,205,223,241,259,277,8,26,44,62,80,98,116,134,...
    152,170,188,206,224,242,260,278,9,27,45,63,81,99,117,135,153,...
    171,189,207,225,243,261,279,10,28,46,64,82,100,118,136,154,172,...
    190,208,226,244,262,280,11,29,47,65,83,101,119,137,155,173,191,...
    209,227,245,263,281,12,30,48,66,84,102,120,138,156,174,192,210,...
    228,246,264,282,13,31,49,67,85,103,121,139,157,175,193,211,229,...
    247,265,283,14,32,50,68,86,104,122,140,158,176,194,212,230,248,...
    266,284,15,33,51,69,87,105,123,141,159,177,195,213,231,249,267,...
    285,16,34,52,70,88,106,124,142,160,178,196,214,232,250,268,286,...
    17,35,53,71,89,107,125,143,161,179,197,215,233,251,269,287,18,...
    36,54,72,90,108,126,144,162,180,198,216,234,252,270,288]);
end
```

```
function [output,rec_signal,parity_bit,data_rate,data_length,tail] = ...
   demodData11(input, start_of_data,H,sent_data_rate,sent_data_length)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% demodData11.m
%
%  Format received bits into OFDM symbols; apply channel estimation;
%  read SIGNAL field; read data bits
%
%   Input:                                                              %
%      input: received bits
%      start_of_data:      index of first bit of CP of SIGNAL symbol
%      H:                  channel frequency response
%      sent_data_rate:     sent data rate
%      sent_data_length:   used as a check of proper header decoding
%   Output:                                                             %
%      output:         received data bits
%      rec_signal:     received SIGNAL field
%      parity_bit:     whether or not parity check passed
%      data_rate:      received data_rate
%      modulation:     received modulation
%      coding_rate:    received coding rate
%      Nsym:           received number of OFDM data symbols
%      data_length:    length of received data in bytes
%
%   Author:     Keith Howland
%   Created:    15 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%==================================%
%make sure received data is divisible by 80
%if not, append zeros
%==================================%
len = length(input(start_of_data:length(input)));
if rem(len,80) ~= 0 %each block should be 80 = 16+64
   pad = 80 - rem(len,80); %if not append zeros;
else pad =0;
end
y1 = [input(start_of_data:length(input)); zeros(pad,1)];

%==================================%
```

```
%serial to parallel conversion
%==================================%
y2=reshape(y1,80,length(y1)/80);      %Reshape data; 64 + 16 = 80;
y3 = y2(17:80,:);         %strip off cyclic prefix
Y64=fft(y3,64);             %bring in to freq domain
Y1 = Y64([2:7,9:21,23:27,39:43,45:57,59:64],:); %remove pilots and nulls

%====================================%
%plot constellation
%====================================%
% yplot=Y1(:);
% h=scatterplot(yplot(1:48),1,0,'+r'); %plot SIGNAL field; BPSK
% hold on;
% scatterplot(yplot(49:end),1,0,'.b',h);%plot data
% title('Constellation of Received Signal and Data fields');
% legend('Signal','Data');
% hold off

%====================================%
%apply channel estimation correction
%====================================%
H1=H([2:7,9:21,23:27,39:43,45:57,59:64]);
for i = 1:size(Y1,2)
   Y1(:,i) = Y1(:,i)./H1;
end
% ynewplot = Y1(:);   %plot corrected channel
% h=scatterplot(ynewplot(1:48),1,0,'+r');
% hold on
% scatterplot(ynewplot(49:end),1,0,'.b',h);
% title('Channel Corrected Constellation');
% legend('Signal','Data');
% hold off

%====================================%
%look at signal field
%====================================%
Y2 = FFTremap11(Y1(:,1));
Y3= signalDemodulate11(Y2,2);%SIGNAL always sent as BPSK
Y4= deinterleaver11(Y3,48,1);
rec_signal= deCode11(Y4,.5)'; %SIGNAL always sent @ 1/2 rate coding
rec_signal= rec_signal + zeros(length(rec_signal),1);  %convert to numbers
[data_rate,modulation,Ncbps,Nbpsc,coding_rate,Nsym,data_length, ...
   parity_bit,tail] = ReadSIGNALField(rec_signal);%read contents of SIGNAL
if data_rate ~= sent_data_rate || data_length ~= sent_data_length ...
      || Nsym+1 ~= size(Y1,2) %if corrupted bits, return all zeros
```

```
    output =0;
    return      %exit out if corrupted signal field
end

%=======================================%
%now read rest of signal at appropriate data rate
%=======================================%
rec_data = Y1(:,2:Nsym+1); %strip off signal symbol
deint_data=zeros(Ncbps,Nsym);
for i = 1: Nsym
    Y2 = FFTremap11(rec_data(:,i));
    Y3= signalDemodulate11(Y2,modulation);
    deint_data(:,i)= deinterleaver11(Y3,Ncbps,Nbpsc);
end
deint_data = deint_data(:); %one stream for decoding and descrambling
decoded_data= deCode11(deint_data,coding_rate);
descrambled_data = descrambler(decoded_data);
output= descrambled_data(17:17+data_length*8-1);   %strip off SERVICE field
output = output';                                  %and tail and pad zeros
```

```
function [output] = descrambler(input)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% descrambler.m
%
%  Unscramble received bits after decoding
%
%   Input:                                                              %
%      input:          scrambled bits
%
%   Output:                                                             %
%      output:         unscrambled bits
%
%   Author:     Keith Howland
%   Created:    29 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


output=zeros(1,length(input));
%set initial state
reg = [1 0 1 1 1 0 1];
for i = 1:length(input)
   new = xor(reg(4),reg(7));
   %shift bits left one
   reg = [new reg([1 2 3 4 5 6])];
   % modulo 2 add output with data bit
   output(i) = xor(new,input(i));
end
```

```
function [output] = FFTremap11(input)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% FFTremap.m
%
%  Changes bit order for input to FFT
%
%   Input:                                                              %
%      input:         input values for FFT
%
%   Output:                                                             %
%      output:        input values for FFT in correct order
%
%   Author:     Keith Howland
%   Created:    29 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

output(1:24) = input(25:48);
output(25:48) = input(1:24);
```

```
function [data_rate,modulation,Ncbps,Nbpsc,coding_rate,Nsym, ...
   data_length,parity_bit,tail] = ReadSIGNALField(signal)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% ReadSIGNALfield.m
%
%   Read the signal fied portion of the PLCP Header to extract: data
%   rate, type of modulation, length of following data, code rate, and
%   number of symbols per subcarrier and data bits per subcarrier
%
%   Input:                                                              %
%      signal: bits that should be SIGNAL Field from frame synch        %
%   Output:                                                             %
%      data rate: data rate at which following data was sent            %
%      modulation: symbol mapping described by M-ary number 2-BPSK,
%         4-QPSK, 16-QAM, 64-QAM
%      Ncbps: number of coded bits per symbol
%      Nbpsc: number of coded bits per subcarrier
%      coding_rate: data bits/ coded bits
%      Nsym: number of OFDM symbols to follow
%      data_length: length of data to follow in bytes
%      tai: sum of six tail bits: should be zero
%
%   Author:     Keith Howland
%   Created:    20 Feb 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%=========================================%
%read out data length from signal field
%=========================================%
data_length = bin2dec(num2str(flipud(signal(6:17))'));

%=========================================%
%read tail
%=========================================%
tail = signal(19:24);%should be six zeros
tail = sum(tail);%should be zero

%=====================================%
% parity calculation of SIGNAL Field
%=====================================%
rec_parity_bit = signal(18);
```

```
sent_parity_bit = rem(sum(signal(1:17)),2);
if rec_parity_bit == sent_parity_bit
   parity_bit = 1;
else
   parity_bit = 0;
end

%==================================%
%Read data rate bits and compare against know codes
%then extract other information
%if no match for rate, set all other fields to zero
%==================================%
rate = signal(1:4);     %get data rate bits
if rate == [1;1;0;1]
   data_rate =6;   %Mbps for data to follow
   Nbpsc =1;       %Number of coded bits per sub carrier
   Ndbps = 24;     %Number of data bits per sub carrier
   Ncbps =48;      %Number of coded bits per OFDM symbol
   modulation = 2;     % 2=BPSK; 4=QPSK; 16=16_QAM 64=64_QAM
   coding_rate = .5;   %convolution encoding rate
elseif rate == [1;1;1;1]
   data_rate = 9;
   Nbpsc =1;
   Ndbps = 36;
   Ncbps =48;
   modulation = 2;
   coding_rate = .75;
elseif rate == [0;1;0;1]
   data_rate = 12;
   Nbpsc =2;
   Ndbps = 48;
   Ncbps =96;
   modulation = 4;
   coding_rate = .5;
elseif rate == [0;1;1;1]
   data_rate = 18;
   Nbpsc =2;
   Ndbps = 72;
   Ncbps =96;
   modulation = 4;
   coding_rate = .75;
elseif rate == [1;0;0;1]
   data_rate = 24;
   Nbpsc =4;
   Ndbps = 96;
```

```
    Ncbps =192;
    modulation = 16;
    coding_rate = .5;
elseif rate == [1;0;1;1]
    data_rate = 36;
    Nbpsc =4;
    Ndbps = 144;
    Ncbps =192;
    modulation = 16;
    coding_rate = .75;
elseif rate == [0;0;0;1]
    data_rate = 48;
    Nbpsc =6;
    Ndbps = 192;
    Ncbps =288;
    modulation = 64;
    coding_rate = .66;
elseif rate == [0;0;1;1]
    data_rate = 54;
    Nbpsc =6;
    Ndbps = 216;
    Ncbps =288;
    modulation = 64;
    coding_rate = .75;
else
    %use data rate = 0 as sign of data rate error
    data_rate=0;modulation=0;Ncbps=0;Nbpsc=0;coding_rate=0;Nsym=0;
    return
end
%=======================================%
%calculate number of OFDM symbols
%=======================================%
Nsym = ceil((16+8*data_length+6)/Ndbps);

%end function
```

```
function [output] = signalDemodulate11(input,mod)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% signalDemodulate11.m
%
%  Map complex valued symbols back to coded bits
%
%   Input:                                                              %
%      sent_data_rate:     complex valued symbols
%      mod:                M-ary number of modulation
%   Output:                                                             %
%      output:             coded bits
%
%   Author:     Keith Howland
%   Created:    29 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%BPSK
if mod ==2
   output = input > 0;
%QPSK
elseif mod ==4
   % Create a demodulator object
   g = modem.qamdemod('M',4,'SymbolOrder','user-defined',...
      'SymbolMapping',[1 0 3 2],'OutputType','bit');
   % Demodulate the signal
   output1 = demodulate(g,sqrt(2)*input);
   output = output1(:);
%16-QAM
elseif mod ==16
   g = modem.qamdemod('M',16, 'SymbolOrder','user-defined',...
      'SymbolMapping',[2 3 1 0 6 7 5 4 14 15 13 12 10 11 9 8],...
      'OutputType','bit');
   output1 = demodulate(g,sqrt(10)*input);
   output = output1(:);
%64-QAM
elseif mod ==64
   g = modem.qamdemod('M',64,'SymbolOrder','user-defined',...
      'SymbolMapping',[4 5 7 6 2 3 1 0 12 13 15 14 10 11 9 8 28 29 31 ...
      30 26 27 25 24 20 21 23 22 18 19 17 16 52 53 55 54 50 51 49 48 ...
      60 61 63 62 58 59 57 56 44 45 47 46 42 43 41 40 36 37 39 38 34 ...
      35 33 32], 'OutputType','bit');  % Create a demodulator object
```

```
    output1 = demodulate(g,sqrt(42)*input);   % Demodulate the signal y.
    output = output1(:);
end
```

```
function [signal_detected,start_of_data,H] = synch11(input,Th)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% synch11.m
%
%  Perform frame synchronization; estimate channel
%
%   Input:                                                              %
%      input:      received bits from channel
%      Th:         decision threshold
%   Output:                                                             %
%      signal_detected:    was the signal detected: 1 - Yes;0 - No;
%      start_of_data:      sample index of start of SIGNAL field
%      H:                  channel frequency response
%
%   Author:     Keith Howland
%   Created:    15 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%===================================%
%load in preambles for cross-correlation
%===================================%
load Receiver/short_time     %time domain of one short preamble symbol
load Receiver/flip_conj_short_preamble  %flipped and complex conjugated
xsp = short_time(1:16);
load Receiver/long_time  %one symbol of long preamble in time domain
load Receiver/flip_conj_long_preamble %above flipped and complex conjugated

%===================================%
%take input and place into blocks of L; L=16 for short preamble
%perform cross-correlation
%look for correlation values above threshold
%===================================%
L=16;n=1;short_ind=zeros(1,10);AGC=0;
short_corr = zeros(1,length(input));
for i = 1: length(input)-2*L;   %L=sample length that is repeated
   b1 = input(i:i+L-1);  %sliding window
   P = xsp' * b1;  %cross correlation with stored preamble
   Ps = P' * P;    %get magnitude
   R2 = (xsp' * xsp);        %power of stored preamble for normalization
   Rs = R2 * R2; %square term so proportional to Ps
```

```
   short_corr(i) = Ps / Rs; %normalize with respect to preamble power
   %Apply Automatic Gain Control to faded signals: see below
%    if AGC == 1
%        short_corr(i) = short_corr(i) * pow_correction;
%    end
   %find peaks above threshold
   if short_corr(i) > Th && n == 1;
      short_ind(n) = i;
      n = n+1;
   elseif short_corr(i) > Th && n < 11  && short_ind(n-1)+ 16 == i
      short_ind(n) = i;
      n = n+1;
%        if n == 5
%           %Automatic Gain Control: Boost faded signals
%           r_pow =(input(i-64:i)' * input(i-64:i))/64;
%           p_pow =(short_time' * short_time)/64;
%           if p_pow > r_pow
%              AGC=1;
%              pow_correction = (p_pow/r_pow);
%           end
%        end
   end
end

%figure;plot(short_corr(400:700));  %plot cross-correlation
%===================================%
%16 samples after last peak starts the long preamble
%===================================%
if n == 11
   start_of_data = short_ind(10) + 16 +160;
   signal_detected=1;
else
   signal_detected=0;H=0;start_of_data=0;
   return
end

%===================================%
%Estimate Channel
%===================================%
% L=16;                    %length of channel set by standard to 16
LP=input(start_of_data -160:start_of_data-1);
LP1=LP(33:96);              %1st Long Preamble
LP2=LP(97:160);          %2nd Long Preabmle
% % y(n) = h(n)*xlp(n) + noise;
% % y= Xh + noise in vector notation
```

```
% c = long_time;r = [long_time(1);long_time(64:-1:64-L+2)]; %column wins
% X = toeplitz(c,r);
% h1 = X\LP1;
% h2 = X\LP2;
% h=0.5*(h1+h2);              %channel impulse response
% figure;plot(h .* conj(h));
% H=fft(h,64); %Frequency response of channel
%
% %==================================%
% %plot channel impulse response
% %==================================%
%
% Y1 = fft(LP1);
% Y2 = fft(LP2);
% X = fft(long_time);
% a = find(round(abs(X))==1);
% X1 = X(a);
% H = (Y1(a)+Y2(a))./(2*X1);
% h1 =ifft(H);
% figure;plot(h1 .* conj(h1));
% % a=1;

xLP=long_time;  % Long Preamble
L=16;  % choose L>16 to account for uncertainty in the estimated beginning of the
preamble
X=toeplitz(xLP, [xLP(1); xLP(64:-1:64-L+2)]);
Xinv=inv(X'*X)*X';
h1=Xinv*LP1;
h2=Xinv*LP2;
h=0.5*(h1+h2);
H=fft(h,64);
%figure;plot(abs(H));xlabel('Frequency');ylabel('Magnitude')
```

```
function [output,cols] = checkBlockSize(input,mod_number)
%check to make sure data fits in uncoded block size
%then find how many blocks of data (cols)

len = length(input);    %get input length

if mod_number == 0
   if mod(len,88) ~= 0       %if input doesn't fit block size -8 for pad bit at end
      input = [input; ones(88 - mod(len,88),1)];  %add ones to end; see 8.3.3.1
   end
   cols = ceil(length(input)/88);    %get number of blocks
   output = reshape(input,88,cols);    %put into matrix for processing
elseif mod_number == 1
   if len ~= 184
      input = [input; ones(184 - mod(len,184),1)];
   end
   cols = length(input)/184;
   output = reshape(input,184,cols);
elseif mod_number == 2
   if  len ~= 280
      input = [input; ones(280 - mod(len,280),1)];
   end
   cols = length(input)/280;
   output = reshape(input,280,cols);
elseif mod_number == 3
   if  len ~= 376
      input = [input; ones(376 - mod(len,376),1)];
   end
   cols = length(input)/376;
   output = reshape(input,376,cols);
elseif mod_number == 4
   if  len ~= 568
      input = [input; ones(568 - mod(len,568),1)];
   end
   cols = length(input)/568;
   output = reshape(input,568,cols);
elseif mod_number == 5
   if  len ~= 760
      input = [input; ones(760 - mod(len,760),1)];
   end
   cols = length(input)/760;
   output = reshape(input,760,cols);
elseif mod_number == 6
   if  len ~= 856
      input = [input; ones(856 - mod(len,856),1)];
```

```
    end
    cols = length(input)/856;
    output = reshape(input,856,cols);
end
```

```
function [output] = ConvolutionCoding16(input,mod_number)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% ConvoultionCoding16.m
%
%  Perform binary convolution encoding on RSecoding output; data is
%  punctured based on modulation rate;gen polys are reverse of 802.11a
%  constraint length is 7
%
%   Input:                                                              %
%      input:          Reed-Solomon encoded bits
%      mod_number:     modulation code number
%   Output:                                                             %
%      output:         binary convolutionally encoded data
%
%   Author:     Keith Howland
%   Created:    20 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%rate 1/2 coder
trellis = poly2trellis(7,[171 133]);    %G1 = 171; G2=133

%now encode and puncture
if mod_number == 0 %rate 1/2 no puncture
   output = convenc(input,trellis);
elseif mod_number ==1 || mod_number ==3        %puncture code for 2/3
   output = convenc(input,trellis,[1 1 0 1]);
elseif mod_number == 5                         %puncture code for 3/4
   output = convenc(input,trellis,[1 1 0 1 1 0]);
elseif mod_number ==2 || mod_number ==4 || mod_number ==6 %puncture code for 5/6
   output = convenc(input,trellis,[1 1 0 1 1 0 0 1 1 0]);
end
```

```
function [signal,fch_bits,data] = ...
   createDownLinkFrame( mod_number, sent_data_length)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% createDownLinkFrame.m
%
%  this function creates the frames from a Base Station
%  in the first burst using the long preamble and creating the appropriate
%  FCH symbol which holds the rate ID for use in decoding
%  CP needs to be made variable
%
%   Input:                                                              %
%      mod_number:        modulation for transmission
%      sent_data_length:  number of bytes in MAC layer data
%
%   Output:                                                             %
%      signal:           time domain sampled signal
%      fch_bits:          PLCP header bits
%      data:             MAC layer data
%
%   Calls:
%      CheckBlockSize.m: make sure right number of input bits for symbols
%      FCH.m: create header
%      Randomizer.m: randomize data
%      RScoding.m: Reed-Solomon encoding
%      ConvolutionCoding16.m: binary convoluion encoding
%      interleaver16.m: interleave data
%      modData16.m: to map bits into symbols: BPSK,QPSK,etc.
%      createSymbol16.m: make 256 length OFDM symbol with pilots and nulls
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Long Preamble first
load long_preamble.mat
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %FCH: encoded at BPSK 1/2 convolution coding only
[fch,fch_bits] = FCH(mod_number,1,1);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% DL Bursts:  assume only one burst
%for now just random binary
data=randint(sent_data_length,1);
[data, cols] = checkBlockSize(data,mod_number);
% cols  % how many cols to screen; can be slow; see progress
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


extended_symbol =zeros(320,cols);    %reserve space for output

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%make the symbols
for i = 1:cols
   %i  %see progress for long input
   random_data=Randomizer(data(:,i),2);
   rs_data=RScoding(random_data,mod_number);
   cc_data=ConvolutionCoding16(rs_data,mod_number);
   interleaved_data=interleaver16(cc_data,mod_number);
   modulated_data = modData16(interleaved_data,mod_number);
   symbol = createSymbol16(modulated_data);
   time_symbol = ifft(symbol,256);
   extended_symbol(:,i) = [time_symbol(193:256) time_symbol];   % add CP
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%add preamble and FCH symbol to data
signal = [long_preamble; fch; extended_symbol(:)];
```

```
function [symbol] = createPilots16(symbol,index)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% createPilots16.m
%
%  %this function creates index for pilot modulation for downlink only
%
%   Input:                                                              %
%      symbol:          OFDM symbol without pilots
%      index:           sample index for pilot
%
%   Output:                                                             %
%      symbol:          OFDM symbol with pilots
%
%   Calls:         None
%
%   Author:    Keith Howland
%   Created:   25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


global reg  %need to retain value through different calls

if index ==0
   %UL initialization
   reg = [1 0 1 0 1 0 1 0 1 0 1];
   %DL initialization
   %reg = [1 1 1 1 1 1 1 1 1 1 1];
end
%w = reg(11); works here for downlink
new = xor(reg(9),reg(11));
%shift bits left one
reg = [new reg(1:10)];
w = reg(11);
w_opp = w==0; %create opposite term

%add pilots for UL
symbol([13 63 113 138 163 188]) = 1-2*w;
symbol([38 88]) = 1 -2*w_opp;
```

```
function [long_preamble] = createPreamble16()
%need to add cyclic prefix variable


%create index for preamble frequency domain sequence
PALL(1,:) = -100:100;
PALL(2,:) = [1 - 1i,1 - 1i,-1 - 1i,1 + 1i,1 - 1i,1 - 1i,-1 + 1i,1 - 1i,1 - 1i,1 - 1i,1 + 1i,-1 -
1i,1 + 1i,1 + 1i,-1 - 1i,1 + 1i,-1 - 1i,-1 - 1i,1 - 1i,-1 + 1i,1 - 1i,1 - 1i,-1 - 1i,1 + 1i,1 - 1i,1 -
1i,-1 + 1i,1 - 1i,1 - 1i,1 - 1i,1 + 1i,-1 - 1i,1 + 1i,1 + 1i,-1 - 1i,1 + 1i,-1 - 1i,-1 - 1i,1 - 1i,-1
+ 1i,1 - 1i,1 - 1i,-1 - 1i,1 + 1i,1 - 1i,1 - 1i,-1 + 1i,1 - 1i,1 - 1i,1 - 1i,1 + 1i,-1 - 1i,1 + 1i,1 +
1i,-1 - 1i,1 + 1i,-1 - 1i,-1 - 1i,1 - 1i,-1 + 1i,1 + 1i,1 + 1i,1 - 1i,-1 + 1i,1 + 1i,1 + 1i,-1 - 1i,1
+ 1i,1 + 1i,1 + 1i,-1 + 1i,1 - 1i,-1 + 1i,-1 + 1i,1 - 1i,-1 + 1i,1 - 1i,1 - 1i,1 + 1i,-1 - 1i,-1 -
1i,-1 - 1i,-1 + 1i,1 - 1i,-1 - 1i,-1 - 1i,1 + 1i,-1 - 1i,-1 - 1i,-1 - 1i,1 - 1i,-1 + 1i,1 - 1i,1 - 1i,-1
+ 1i,1 - 1i,-1 + 1i,-1 + 1i,-1 - 1i,1 + 1i,0 + 0i,-1 - 1i,1 + 1i,-1 + 1i,-1 + 1i,-1 - 1i,1 + 1i,1 +
1i,1 + 1i,-1 - 1i,1 + 1i,1 - 1i,1 - 1i,1 - 1i,-1 + 1i,-1 + 1i,-1 + 1i,-1 + 1i,1 - 1i,-1 - 1i,-1 - 1i,-
1 + 1i,1 - 1i,1 + 1i,1 + 1i,-1 + 1i,1 - 1i,1 - 1i,1 - 1i,-1 + 1i,1 - 1i,-1 - 1i,-1 - 1i,-1 - 1i,1 +
1i,1 + 1i,1 + 1i,1 + 1i,-1 - 1i,-1 + 1i,-1 + 1i,1 + 1i,-1 - 1i,1 - 1i,1 - 1i,1 + 1i,-1 - 1i,-1 - 1i,-
1 - 1i,1 + 1i,-1 - 1i,-1 + 1i,-1 + 1i,-1 + 1i,1 - 1i,1 - 1i,1 - 1i,1 - 1i,-1 + 1i,1 + 1i,1 + 1i,-1 -
1i,1 + 1i,-1 + 1i,-1 + 1i,-1 - 1i,1 + 1i,1 + 1i,1 + 1i,-1 - 1i,1 + 1i,1 - 1i,1 - 1i,1 - 1i,-1 + 1i,-
1 + 1i,-1 + 1i,-1 + 1i,1 - 1i,-1 - 1i,-1 - 1i,1 - 1i,-1 + 1i,-1 - 1i,-1 - 1i,1 - 1i,-1 + 1i,-1 + 1i,-1
+ 1i,1 - 1i,-1 + 1i,1 + 1i,1 + 1i,1 + 1i,-1 - 1i,-1 - 1i,-1 - 1i,-1 - 1i,1 + 1i,1 - 1i,1 - 1i];

%make first preamble symbol
b=mod(-100:100,4); %remember based off -100:100
c=find(b);
DL_preamble_1=PALL;
DL_preamble_1(2,c)=0;    %only mod(k,4) =0 have value
DL_preamble_1 = conj(DL_preamble_1(2,:)).';
DL_preamble_1 = 2 * DL_preamble_1;
preamble_DL_1 = ifft(DL_preamble_1,256);
preamble_DL_1_CP = [preamble_DL_1(193:end) preamble_DL_1];   %add cyclic prefix

%make second preamble symbol for DL
%this is also the preamble for UL known as short preamble
d=mod(-100:100,2); %remember based off -100:100
f=find(d);
DL_preamble_2=PALL;
DL_preamble_2(2,f)=0;
symbol_2=sqrt(2)*DL_preamble_2(2,:);

preamble_DL_2 = ifft(symbol_2,256);
preamble_DL_2_CP = [preamble_DL_2(193:end) preamble_DL_2];    %add cyclic
prefix

long_preamble = [preamble_DL_1_CP preamble_DL_2_CP];
```

```
function [OFDM_symbol] = createSymbol16(modulated_data)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% createSymbol16.m
%
%  %this function creates index for pilot modulation for downlink only
%
%   Input:                                                              %
%      modulated_data:     input data for OFDM symbol
%
%   Output:                                                             %
%      OFDM_symbol:            OFDM symbol with pilots and nulls
%
%   Calls:
%      createPilots16.m: to add piltos
%      IFFTmap16.m: to arrane order of samples for FFT
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%create frame of symbols
symbol(1,:) = [-100:-1 1:100];
symbol(2,:) = zeros(1,200); %start with all nulls

%add data when using all 16 subcarriers
symbol(2,([1:12,14:37,39:62,64:87,89:112,114:137,139:162,164:187,189:200]))=modula
ted_data;

%insert pilots
%simulate creating preamble for testing
for i = 0:1
   w = createPilots16(symbol(2,:),i);
end
symbol = createPilots16(symbol(2,:),2);    %dummy parameter right now

OFDM_symbol = IFFTmap16(symbol);
```

```
function [output,fch_bits] = FCH(mod_number,preamble,DIUC)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% FCH.m
%
%  create the PLCP header and encode at 1/2 rate with BPSK modulation
%
%   Input:                                                              %
%      mod_number:        modulation for transmission
%      preamble:          preamble sample values
%      DIUC:              information regarding frame
%
%   Output:                                                             %
%      output:            OFDM header symbol
%      fch_bits:          PLCP header bits
%
%   Calls:
%      Randomizer.m: randomize data
%      RScoding.m: Reed-Solomon encoding
%      ConvolutionCoding16.m: binary convoluion encoding
%      interleaver16.m: interleave data
%      modData16.m: to map bits into symbols: BPSK,QPSK,etc.
%      createSymbol16.m: make 256 length OFDM symbol with pilots and nulls
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%convert mod_number to binary
if mod_number == 0
   mod_number = [0 0 0 0];
elseif mod_number ==1
   mod_number = [0 0 0 1];
elseif mod_number == 2
   mod_number = [0 0 1 0];
elseif mod_number == 3
   mod_number = [0 0 1 1];
elseif mod_number == 4
   mod_number = [0 1 0 0];
elseif mod_number == 5
   mod_number = [0 1 0 1];
```

```
elseif mod_number == 6
   mod_number = [0 1 1 0];
end

base_ID = [0 0 0 1];    %base station ID
frame_number = [0 0 0 1];   % see Table 358 of standard
config_cc = [0 0 0 1];      %see 6.3.2.3.1
reserve = [0 0 0 0];    %reserved bits
bits=[];
for i = 0:3
   if i == 0
      Rate_DIUC = mod_number; %0=BPSK,.5; 1=QPSK,.5; 2=QPSK,.75; etc.
   else
      Rate_DIUC = [0 0 0 1];
   end
   if DIUC ~=0
      if preamble ==1
         pre = 1;
      else
         pre=0;
      end
      len = [zeros(1,10) 1];  % #of OFDM symbols in burst
      DIUC = [pre len];
   else
      start_time = [zeros(1,11) 1];
      DIUC = start_time;
   end
   bits = [bits Rate_DIUC DIUC];
end

fch_bits_no_HCS = [base_ID frame_number config_cc reserve bits];

hcs = HCS(fch_bits_no_HCS);
fch_bits = [fch_bits_no_HCS hcs];

%create PLCP Header
rand_fch = Randomizer(fch_bits,2);      %randomize; burst 2 for UL initial.
cc_fch = ConvolutionCoding16(rand_fch,0); %1/2 rate convolution coding
int_fch = interleaver16(cc_fch,0);       %interleaving
mod_fch = modData16(int_fch,0);          %BPSK modulation
fch_sym = createSymbol16(mod_fch);       %add pilots and nulls
time_fch = ifft(fch_sym,256).';         %bring into time domain
output = [time_fch(193:256); time_fch]; %add cyclic prefix; fixed at 64 right now
```

```
function [reg] = HCS(input)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% HCS.m
%
%  cyclic redundancy check for header bits
%  g(D) = D^8 + D^2 + D + 1
%
%   Input:                                                              %
%      inputs:            header bits
%
%   Output:                                                             %
%     reg:               CRC bits
%
%   Calls:                None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


reg = zeros(1,8);   %set initial state
for i = 1:length(input)
   reg8 = xor(reg(1),input(i));   %my reg8 = x(0); in other words reversed
   reg7 = xor(reg(8),reg8);
   reg6 = xor(reg(7),reg8);
   %shift bits left one
   reg = [reg(2:6) reg6 reg7 reg8];
end
```

```
function [IFFT_symbol] = IFFTmap16(input)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% IFFTmap16.m
%
%  change order of input for IFFT; put zeros in middle;
%
%   Input:                                                              %
%      inputs:            samples
%
%   Output:                                                             %
%     IFFT_symbol:        order samples
%
%   Calls:                None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


IFFT_symbol(1) = 0;
IFFT_symbol(2:101) = input(101:200);
IFFT_symbol(102:156)= zeros(1,55);
IFFT_symbol(157:256)= input(1:100);
```

```
function [output] = interleaver16(bits,mod_number)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% interleaver16.m
%
%  interleave bits based on modulation type; per page 17 of PHY section of
%  802.11a standard for 16 subchannels
%
%   Input:                                                              %
%      bits:              bits for interleaving
%      mod_number:        modulation type
%
%   Output:                                                             %
%      output:            interleaved bits
%
%   Calls:                None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if mod_number == 0
   first =bits([1,13,25,37,49,61,73,85,97,109,121,133,145,157,169,181,...
      2,14,26,38,50,62,74,86,98,110,122,134,146,158,170,182,3,15,27,...
      39,51,63,75,87,99,111,123,135,147,159,171,183,4,16,28,40,52,64,...
      76,88,100,112,124,136,148,160,172,184,5,17,29,41,53,65,77,89,...
      101,113,125,137,149,161,173,185,6,18,30,42,54,66,78,90,102,114,...
      126,138,150,162,174,186,7,19,31,43,55,67,79,91,103,115,127,139,...
      151,163,175,187,8,20,32,44,56,68,80,92,104,116,128,140,152,164,...
      176,188,9,21,33,45,57,69,81,93,105,117,129,141,153,165,177,189,...
      10,22,34,46,58,70,82,94,106,118,130,142,154,166,178,190,11,23,...
      35,47,59,71,83,95,107,119,131,143,155,167,179,191,12,24,36,48,...
      60,72,84,96,108,120,132,144,156,168,180,192]);
   output = first([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
      21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,...
      42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,...
      63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,...
      84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,...
      103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,...
      118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,...
      133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,...
      148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,...
```

```
        163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,...
        178,179,180,181,182,183,184,185,186,187,188,189,190,191,192]);
elseif mod_number == 1 || mod_number == 2
    first = bits([1,13,25,37,49,61,73,85,97,109,121,133,145,157,169,...
        181,193,205,217,229,241,253,265,277,289,301,313,325,337,349,...
        361,373,2,14,26,38,50,62,74,86,98,110,122,134,146,158,170,182,...
        194,206,218,230,242,254,266,278,290,302,314,326,338,350,362,...
        374,3,15,27,39,51,63,75,87,99,111,123,135,147,159,171,183,195,...
        207,219,231,243,255,267,279,291,303,315,327,339,351,363,375,4,...
        16,28,40,52,64,76,88,100,112,124,136,148,160,172,184,196,208,...
        220,232,244,256,268,280,292,304,316,328,340,352,364,376,5,17,...
        29,41,53,65,77,89,101,113,125,137,149,161,173,185,197,209,221,...
        233,245,257,269,281,293,305,317,329,341,353,365,377,6,18,30,42,...
        54,66,78,90,102,114,126,138,150,162,174,186,198,210,222,234,...
        246,258,270,282,294,306,318,330,342,354,366,378,7,19,31,43,55,...
        67,79,91,103,115,127,139,151,163,175,187,199,211,223,235,247,...
        259,271,283,295,307,319,331,343,355,367,379,8,20,32,44,56,68,...
        80,92,104,116,128,140,152,164,176,188,200,212,224,236,248,260,...
        272,284,296,308,320,332,344,356,368,380,9,21,33,45,57,69,81,93,...
        105,117,129,141,153,165,177,189,201,213,225,237,249,261,273,...
        285,297,309,321,333,345,357,369,381,10,22,34,46,58,70,82,94,...
        106,118,130,142,154,166,178,190,202,214,226,238,250,262,274,...
        286,298,310,322,334,346,358,370,382,11,23,35,47,59,71,83,95,...
        107,119,131,143,155,167,179,191,203,215,227,239,251,263,275,...
        287,299,311,323,335,347,359,371,383,12,24,36,48,60,72,84,96,...
        108,120,132,144,156,168,180,192,204,216,228,240,252,264,276,...
        288,300,312,324,336,348,360,372,384]);
    output = first([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
        21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,...
        42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,...
        63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,...
        84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,...
        103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,...
        118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,...
        133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,...
        148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,...
        163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,...
        178,179,180,181,182,183,184,185,186,187,188,189,190,191,192,...
        193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,...
        208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,...
        223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,...
        238,239,240,241,242,243,244,245,246,247,248,249,250,251,252,...
        253,254,255,256,257,258,259,260,261,262,263,264,265,266,267,...
        268,269,270,271,272,273,274,275,276,277,278,279,280,281,282,...
        283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,...
```

```
        298,299,300,301,302,303,304,305,306,307,308,309,310,311,312,...
        313,314,315,316,317,318,319,320,321,322,323,324,325,326,327,...
        328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,...
        343,344,345,346,347,348,349,350,351,352,353,354,355,356,357,...
        358,359,360,361,362,363,364,365,366,367,368,369,370,371,372,...
        373,374,375,376,377,378,379,380,381,382,383,384]);
elseif mod_number ==3 || mod_number ==4
    first = bits([1,13,25,37,49,61,73,85,97,109,121,133,145,157,169,181,...
        193,205,217,229,241,253,265,277,289,301,313,325,337,349,361,373,...
        385,397,409,421,433,445,457,469,481,493,505,517,529,541,553,565,...
        577,589,601,613,625,637,649,661,673,685,697,709,721,733,745,757,...
        2,14,26,38,50,62,74,86,98,110,122,134,146,158,170,182,194,206,...
        218,230,242,254,266,278,290,302,314,326,338,350,362,374,386,...
        398,410,422,434,446,458,470,482,494,506,518,530,542,554,566,578,...
        590,602,614,626,638,650,662,674,686,698,710,722,734,746,758,3,...
        15,27,39,51,63,75,87,99,111,123,135,147,159,171,183,195,207,219,...
        231,243,255,267,279,291,303,315,327,339,351,363,375,387,399,411,...
        423,435,447,459,471,483,495,507,519,531,543,555,567,579,591,603,...
        615,627,639,651,663,675,687,699,711,723,735,747,759,4,16,28,40,...
        52,64,76,88,100,112,124,136,148,160,172,184,196,208,220,232,244,...
        256,268,280,292,304,316,328,340,352,364,376,388,400,412,424,436,...
        448,460,472,484,496,508,520,532,544,556,568,580,592,604,616,628,...
        640,652,664,676,688,700,712,724,736,748,760,5,17,29,41,53,65,77,...
        89,101,113,125,137,149,161,173,185,197,209,221,233,245,257,269,...
        281,293,305,317,329,341,353,365,377,389,401,413,425,437,449,461,...
        473,485,497,509,521,533,545,557,569,581,593,605,617,629,641,653,...
        665,677,689,701,713,725,737,749,761,6,18,30,42,54,66,78,90,102,...
        114,126,138,150,162,174,186,198,210,222,234,246,258,270,282,294,...
        306,318,330,342,354,366,378,390,402,414,426,438,450,462,474,486,...
        498,510,522,534,546,558,570,582,594,606,618,630,642,654,666,678,...
        690,702,714,726,738,750,762,7,19,31,43,55,67,79,91,103,115,127,...
        139,151,163,175,187,199,211,223,235,247,259,271,283,295,307,319,...
        331,343,355,367,379,391,403,415,427,439,451,463,475,487,499,511,...
        523,535,547,559,571,583,595,607,619,631,643,655,667,679,691,703,...
        715,727,739,751,763,8,20,32,44,56,68,80,92,104,116,128,140,152,...
        164,176,188,200,212,224,236,248,260,272,284,296,308,320,332,344,...
        356,368,380,392,404,416,428,440,452,464,476,488,500,512,524,536,...
        548,560,572,584,596,608,620,632,644,656,668,680,692,704,716,728,...
        740,752,764,9,21,33,45,57,69,81,93,105,117,129,141,153,165,177,...
        189,201,213,225,237,249,261,273,285,297,309,321,333,345,357,369,...
        381,393,405,417,429,441,453,465,477,489,501,513,525,537,549,561,...
        573,585,597,609,621,633,645,657,669,681,693,705,717,729,741,753,...
        765,10,22,34,46,58,70,82,94,106,118,130,142,154,166,178,190,202,...
        214,226,238,250,262,274,286,298,310,322,334,346,358,370,382,394,...
        406,418,430,442,454,466,478,490,502,514,526,538,550,562,574,586,...
```

```
    598,610,622,634,646,658,670,682,694,706,718,730,742,754,766,11,...
    23,35,47,59,71,83,95,107,119,131,143,155,167,179,191,203,215,...
    227,239,251,263,275,287,299,311,323,335,347,359,371,383,395,407,...
    419,431,443,455,467,479,491,503,515,527,539,551,563,575,587,599,...
    611,623,635,647,659,671,683,695,707,719,731,743,755,767,12,24,...
    36,48,60,72,84,96,108,120,132,144,156,168,180,192,204,216,228,...
    240,252,264,276,288,300,312,324,336,348,360,372,384,396,408,420,...
    432,444,456,468,480,492,504,516,528,540,552,564,576,588,600,612,...
    624,636,648,660,672,684,696,708,720,732,744,756,768]);
output= first([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
    21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,...
    42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,...
    63,64,66,65,68,67,70,69,72,71,74,73,76,75,78,77,80,79,82,81,84,...
    83,86,85,88,87,90,89,92,91,94,93,96,95,98,97,100,99,102,101,104,...
    103,106,105,108,107,110,109,112,111,114,113,116,115,118,117,120,...
    119,122,121,124,123,126,125,128,127,129,130,131,132,133,134,135,...
    136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,...
    152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,...
    168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,...
    184,185,186,187,188,189,190,191,192,194,193,196,195,198,197,200,...
    199,202,201,204,203,206,205,208,207,210,209,212,211,214,213,216,...
    215,218,217,220,219,222,221,224,223,226,225,228,227,230,229,232,...
    231,234,233,236,235,238,237,240,239,242,241,244,243,246,245,248,...
    247,250,249,252,251,254,253,256,255,257,258,259,260,261,262,263,...
    264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,...
    280,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,...
    296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,...
    312,313,314,315,316,317,318,319,320,322,321,324,323,326,325,328,...
    327,330,329,332,331,334,333,336,335,338,337,340,339,342,341,344,...
    343,346,345,348,347,350,349,352,351,354,353,356,355,358,357,360,...
    359,362,361,364,363,366,365,368,367,370,369,372,371,374,373,376,...
    375,378,377,380,379,382,381,384,383,385,386,387,388,389,390,391,...
    392,393,394,395,396,397,398,399,400,401,402,403,404,405,406,407,...
    408,409,410,411,412,413,414,415,416,417,418,419,420,421,422,423,...
    424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,...
    440,441,442,443,444,445,446,447,448,450,449,452,451,454,453,456,...
    455,458,457,460,459,462,461,464,463,466,465,468,467,470,469,472,...
    471,474,473,476,475,478,477,480,479,482,481,484,483,486,485,488,...
    487,490,489,492,491,494,493,496,495,498,497,500,499,502,501,504,...
    503,506,505,508,507,510,509,512,511,513,514,515,516,517,518,519,...
    520,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,...
    536,537,538,539,540,541,542,543,544,545,546,547,548,549,550,551,...
    552,553,554,555,556,557,558,559,560,561,562,563,564,565,566,567,...
    568,569,570,571,572,573,574,575,576,578,577,580,579,582,581,584,...
    583,586,585,588,587,590,589,592,591,594,593,596,595,598,597,600,...
```

```
        599,602,601,604,603,606,605,608,607,610,609,612,611,614,613,616,...
        615,618,617,620,619,622,621,624,623,626,625,628,627,630,629,632,...
        631,634,633,636,635,638,637,640,639,641,642,643,644,645,646,647,...
        648,649,650,651,652,653,654,655,656,657,658,659,660,661,662,663,...
        664,665,666,667,668,669,670,671,672,673,674,675,676,677,678,679,...
        680,681,682,683,684,685,686,687,688,689,690,691,692,693,694,695,...
        696,697,698,699,700,701,702,703,704,706,705,708,707,710,709,712,...
        711,714,713,716,715,718,717,720,719,722,721,724,723,726,725,728,...
        727,730,729,732,731,734,733,736,735,738,737,740,739,742,741,744,...
        743,746,745,748,747,750,749,752,751,754,753,756,755,758,757,760,...
        759,762,761,764,763,766,765,768,767]);
elseif mod_number == 5 || mod_number ==6
    first = bits([1,13,25,37,49,61,73,85,97,109,121,133,145,157,169,181,...
        193,205,217,229,241,253,265,277,289,301,313,325,337,349,361,373,...
        385,397,409,421,433,445,457,469,481,493,505,517,529,541,553,565,...
        577,589,601,613,625,637,649,661,673,685,697,709,721,733,745,757,...
        769,781,793,805,817,829,841,853,865,877,889,901,913,925,937,949,...
        961,973,985,997,1009,1021,1033,1045,1057,1069,1081,1093,1105,...
        1117,1129,1141,2,14,26,38,50,62,74,86,98,110,122,134,146,158,...
        170,182,194,206,218,230,242,254,266,278,290,302,314,326,338,350,...
        362,374,386,398,410,422,434,446,458,470,482,494,506,518,530,542,...
        554,566,578,590,602,614,626,638,650,662,674,686,698,710,722,734,...
        746,758,770,782,794,806,818,830,842,854,866,878,890,902,914,926,...
        938,950,962,974,986,998,1010,1022,1034,1046,1058,1070,1082,1094,...
        1106,1118,1130,1142,3,15,27,39,51,63,75,87,99,111,123,135,147,...
        159,171,183,195,207,219,231,243,255,267,279,291,303,315,327,339,...
        351,363,375,387,399,411,423,435,447,459,471,483,495,507,519,531,...
        543,555,567,579,591,603,615,627,639,651,663,675,687,699,711,723,...
        735,747,759,771,783,795,807,819,831,843,855,867,879,891,903,915,...
        927,939,951,963,975,987,999,1011,1023,1035,1047,1059,1071,1083,...
        1095,1107,1119,1131,1143,4,16,28,40,52,64,76,88,100,112,124,136,...
        148,160,172,184,196,208,220,232,244,256,268,280,292,304,316,328,...
        340,352,364,376,388,400,412,424,436,448,460,472,484,496,508,520,...
        532,544,556,568,580,592,604,616,628,640,652,664,676,688,700,712,...
        724,736,748,760,772,784,796,808,820,832,844,856,868,880,892,904,...
        916,928,940,952,964,976,988,1000,1012,1024,1036,1048,1060,1072,...
        1084,1096,1108,1120,1132,1144,5,17,29,41,53,65,77,89,101,113,...
        125,137,149,161,173,185,197,209,221,233,245,257,269,281,293,305,...
        317,329,341,353,365,377,389,401,413,425,437,449,461,473,485,497,...
        509,521,533,545,557,569,581,593,605,617,629,641,653,665,677,689,...
        701,713,725,737,749,761,773,785,797,809,821,833,845,857,869,881,...
        893,905,917,929,941,953,965,977,989,1001,1013,1025,1037,1049,...
        1061,1073,1085,1097,1109,1121,1133,1145,6,18,30,42,54,66,78,90,...
        102,114,126,138,150,162,174,186,198,210,222,234,246,258,270,282,...
        294,306,318,330,342,354,366,378,390,402,414,426,438,450,462,474,...
```

```
    486,498,510,522,534,546,558,570,582,594,606,618,630,642,654,666,...
    678,690,702,714,726,738,750,762,774,786,798,810,822,834,846,858,...
    870,882,894,906,918,930,942,954,966,978,990,1002,1014,1026,1038,...
    1050,1062,1074,1086,1098,1110,1122,1134,1146,7,19,31,43,55,67,...
    79,91,103,115,127,139,151,163,175,187,199,211,223,235,247,259,...
    271,283,295,307,319,331,343,355,367,379,391,403,415,427,439,451,...
    463,475,487,499,511,523,535,547,559,571,583,595,607,619,631,643,...
    655,667,679,691,703,715,727,739,751,763,775,787,799,811,823,835,...
    847,859,871,883,895,907,919,931,943,955,967,979,991,1003,1015,...
    1027,1039,1051,1063,1075,1087,1099,1111,1123,1135,1147,8,20,32,...
    44,56,68,80,92,104,116,128,140,152,164,176,188,200,212,224,236,...
    248,260,272,284,296,308,320,332,344,356,368,380,392,404,416,428,...
    440,452,464,476,488,500,512,524,536,548,560,572,584,596,608,620,...
    632,644,656,668,680,692,704,716,728,740,752,764,776,788,800,812,...
    824,836,848,860,872,884,896,908,920,932,944,956,968,980,992,...
    1004,1016,1028,1040,1052,1064,1076,1088,1100,1112,1124,1136,...
    1148,9,21,33,45,57,69,81,93,105,117,129,141,153,165,177,189,201,...
    213,225,237,249,261,273,285,297,309,321,333,345,357,369,381,393,...
    405,417,429,441,453,465,477,489,501,513,525,537,549,561,573,585,...
    597,609,621,633,645,657,669,681,693,705,717,729,741,753,765,777,...
    789,801,813,825,837,849,861,873,885,897,909,921,933,945,957,969,...
    981,993,1005,1017,1029,1041,1053,1065,1077,1089,1101,1113,1125,...
    1137,1149,10,22,34,46,58,70,82,94,106,118,130,142,154,166,178,...
    190,202,214,226,238,250,262,274,286,298,310,322,334,346,358,370,...
    382,394,406,418,430,442,454,466,478,490,502,514,526,538,550,562,...
    574,586,598,610,622,634,646,658,670,682,694,706,718,730,742,754,...
    766,778,790,802,814,826,838,850,862,874,886,898,910,922,934,946,...
    958,970,982,994,1006,1018,1030,1042,1054,1066,1078,1090,1102,...
    1114,1126,1138,1150,11,23,35,47,59,71,83,95,107,119,131,143,155,...
    167,179,191,203,215,227,239,251,263,275,287,299,311,323,335,347,...
    359,371,383,395,407,419,431,443,455,467,479,491,503,515,527,539,...
    551,563,575,587,599,611,623,635,647,659,671,683,695,707,719,731,...
    743,755,767,779,791,803,815,827,839,851,863,875,887,899,911,923,...
    935,947,959,971,983,995,1007,1019,1031,1043,1055,1067,1079,1091,...
    1103,1115,1127,1139,1151,12,24,36,48,60,72,84,96,108,120,132,...
    144,156,168,180,192,204,216,228,240,252,264,276,288,300,312,324,...
    336,348,360,372,384,396,408,420,432,444,456,468,480,492,504,516,...
    528,540,552,564,576,588,600,612,624,636,648,660,672,684,696,708,...
    720,732,744,756,768,780,792,804,816,828,840,852,864,876,888,900,...
    912,924,936,948,960,972,984,996,1008,1020,1032,1044,1056,1068,...
    1080,1092,1104,1116,1128,1140,1152]);
output = first([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
    21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,...
    42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,...
    63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,...
```

84,85,86,87,88,89,90,91,92,93,94,95,96,98,99,97,101,102,100,104,...
105,103,107,108,106,110,111,109,113,114,112,116,117,115,119,120,...
118,122,123,121,125,126,124,128,129,127,131,132,130,134,135,133,...
137,138,136,140,141,139,143,144,142,146,147,145,149,150,148,152,...
153,151,155,156,154,158,159,157,161,162,160,164,165,163,167,168,...
166,170,171,169,173,174,172,176,177,175,179,180,178,182,183,181,...
185,186,184,188,189,187,191,192,190,195,193,194,198,196,197,201,...
199,200,204,202,203,207,205,206,210,208,209,213,211,212,216,214,...
215,219,217,218,222,220,221,225,223,224,228,226,227,231,229,230,...
234,232,233,237,235,236,240,238,239,243,241,242,246,244,245,249,...
247,248,252,250,251,255,253,254,258,256,257,261,259,260,264,262,...
263,267,265,266,270,268,269,273,271,272,276,274,275,279,277,278,...
282,280,281,285,283,284,288,286,287,289,290,291,292,293,294,295,...
296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,...
312,313,314,315,316,317,318,319,320,321,322,323,324,325,326,327,...
328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,...
344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,...
360,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,...
376,377,378,379,380,381,382,383,384,386,387,385,389,390,388,392,...
393,391,395,396,394,398,399,397,401,402,400,404,405,403,407,408,...
406,410,411,409,413,414,412,416,417,415,419,420,418,422,423,421,...
425,426,424,428,429,427,431,432,430,434,435,433,437,438,436,440,...
441,439,443,444,442,446,447,445,449,450,448,452,453,451,455,456,...
454,458,459,457,461,462,460,464,465,463,467,468,466,470,471,469,...
473,474,472,476,477,475,479,480,478,483,481,482,486,484,485,489,...
487,488,492,490,491,495,493,494,498,496,497,501,499,500,504,502,...
503,507,505,506,510,508,509,513,511,512,516,514,515,519,517,518,...
522,520,521,525,523,524,528,526,527,531,529,530,534,532,533,537,...
535,536,540,538,539,543,541,542,546,544,545,549,547,548,552,550,...
551,555,553,554,558,556,557,561,559,560,564,562,563,567,565,566,...
570,568,569,573,571,572,576,574,575,577,578,579,580,581,582,583,...
584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,...
600,601,602,603,604,605,606,607,608,609,610,611,612,613,614,615,...
616,617,618,619,620,621,622,623,624,625,626,627,628,629,630,631,...
632,633,634,635,636,637,638,639,640,641,642,643,644,645,646,647,...
648,649,650,651,652,653,654,655,656,657,658,659,660,661,662,663,...
664,665,666,667,668,669,670,671,672,674,675,673,677,678,676,680,...
681,679,683,684,682,686,687,685,689,690,688,692,693,691,695,696,...
694,698,699,697,701,702,700,704,705,703,707,708,706,710,711,709,...
713,714,712,716,717,715,719,720,718,722,723,721,725,726,724,728,...
729,727,731,732,730,734,735,733,737,738,736,740,741,739,743,744,...
742,746,747,745,749,750,748,752,753,751,755,756,754,758,759,757,...
761,762,760,764,765,763,767,768,766,771,769,770,774,772,773,777,...
775,776,780,778,779,783,781,782,786,784,785,789,787,788,792,790,...
791,795,793,794,798,796,797,801,799,800,804,802,803,807,805,806,...

```
    810,808,809,813,811,812,816,814,815,819,817,818,822,820,821,825,...
    823,824,828,826,827,831,829,830,834,832,833,837,835,836,840,838,...
    839,843,841,842,846,844,845,849,847,848,852,850,851,855,853,854,...
    858,856,857,861,859,860,864,862,863,865,866,867,868,869,870,871,...
    872,873,874,875,876,877,878,879,880,881,882,883,884,885,886,887,...
    888,889,890,891,892,893,894,895,896,897,898,899,900,901,902,903,...
    904,905,906,907,908,909,910,911,912,913,914,915,916,917,918,919,...
    920,921,922,923,924,925,926,927,928,929,930,931,932,933,934,935,...
    936,937,938,939,940,941,942,943,944,945,946,947,948,949,950,951,...
    952,953,954,955,956,957,958,959,960,962,963,961,965,966,964,968,...
    969,967,971,972,970,974,975,973,977,978,976,980,981,979,983,984,...
    982,986,987,985,989,990,988,992,993,991,995,996,994,998,999,997,...
    1001,1002,1000,1004,1005,1003,1007,1008,1006,1010,1011,1009,...
    1013,1014,1012,1016,1017,1015,1019,1020,1018,1022,1023,1021,...
    1025,1026,1024,1028,1029,1027,1031,1032,1030,1034,1035,1033,...
    1037,1038,1036,1040,1041,1039,1043,1044,1042,1046,1047,1045,...
    1049,1050,1048,1052,1053,1051,1055,1056,1054,1059,1057,1058,...
    1062,1060,1061,1065,1063,1064,1068,1066,1067,1071,1069,1070,...
    1074,1072,1073,1077,1075,1076,1080,1078,1079,1083,1081,1082,...
    1086,1084,1085,1089,1087,1088,1092,1090,1091,1095,1093,1094,...
    1098,1096,1097,1101,1099,1100,1104,1102,1103,1107,1105,1106,...
    1110,1108,1109,1113,1111,1112,1116,1114,1115,1119,1117,1118,...
    1122,1120,1121,1125,1123,1124,1128,1126,1127,1131,1129,1130,...
    1134,1132,1133,1137,1135,1136,1140,1138,1139,1143,1141,1142,...
    1146,1144,1145,1149,1147,1148,1152,1150,1151]);
end
```

```
function [output] = Randomizer(input,burst_number)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% Randomizer.m
%
%  this function randomizes the data per the 802.16e standard
%  this is for the downlink only
%  input data is to enter MSB first.
%
%   Input:                                                              %
%      input:            MAC layer data bits
%      burst_number:      number of burst in transmission
%
%   Output:                                                             %
%     output:            scrambled bits
%
%   Calls:               None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%for testing; further implementation will make this variable
BSID = [0 0 0 1];
UIUC = [0 1 1 1];
Frame_number = [0 0 0 1];

%if start of frame set initial state
if burst_number ==1
   reg = [1 0 0 1 0 1 0 1 0 0 0 0 0 0 0]';
else
   reg = [BSID 1 1 UIUC 1 Frame_number]';  %initializer for UL also
end

output=zeros(1,length(input));
for i = 1:length(input)
   new = xor(reg(14),reg(15));
   reg = [new; reg(1:14)];     %shift bits left one
   output(i) = xor(new,input(i));    % modulo 2 add output with data bit
end

output = [output zeros(1,8)]; %add pad byte of zeros per standard
```

```
function [output]= RScoding(input,mod_number)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% RScoding.m
%
%  Perform shortened Reed-Solomon encoding; shortening depends on
%  modulation for transmission; bits must first be converted to an 8-bit
%  Galois field; then after encoding converted back to binary;
%  very slow code;
%
%   Input:                                                              %
%      input:         radomized data
%      mod_number:     modulation code number
%   Output:                                                             %
%      output:        shortended Reed-Solomon coded data
%
%   Author:     Keith Howland
%   Created:    20 Apr 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%=========================================================%
%need t for shortened Reed Solomon coding to know what redundant bits to
      %discard
if mod_number == 0
   output = input;     %no RS coding here
   return
elseif mod_number ==1
   t=4;
elseif mod_number ==2
   t=2;
elseif mod_number ==3
   t = 8;
elseif mod_number ==4
   t = 4;
elseif mod_number == 5
   t = 6;
else
   t = 6;
end
%=========================================================%
```

```
%input to Galois field must be integer 1-2^m-1;m=8;
input=make8Bit(input);

%Create a Galois array in GF(2^8).
msg = gf(input,8);

len = length(msg);
%determine zeros prefix for shortened RS code
zero_pad = 239 - length(input)+1;
msg_pad = [zeros(1,zero_pad-1) msg];   %prefix zeros
% the key is the zero at the end to specify RS generation polynomial;
% has to due with how Matlab uses the employs the generaor polynomial
gen=rsgenpoly(255,239,285,0);
%Encode the information symbols;very slow function!!!
rscoded = rsenc(msg_pad,255,239,gen);
 %get array in usable form
rs_array = rscoded.x;
%remove extra zeros
data = rs_array(zero_pad:zero_pad + len-1);
 %remove extra redundant bits
redundant = rs_array(zero_pad +len: zero_pad +len + 2*t-1);
%redundant bits sent first
rs_array = [redundant data];
%=========================================%

%=====================================%
%convert to binary
rs_bin_8=zeros(1,8);
rs_out=[];
for i = 1:length(rs_array)
   rs_char = dec2bin(rs_array(i),8);
      for j=1:8
         rs_bin_8(j) =str2double(rs_char(j));
         rs_out = [rs_out rs_bin_8(j)];
      end
end
%=====================================%
%prepare output
output=rs_out';
```

```
function [reg] = CRC(input)
%this function scrambles the data bits as per page 18 of 802.11b-1999

reg = zeros(1,8);   %set initial state
for i = 1:length(input)
   reg8 = xor(reg(1),input(i));   %my reg16 = X(0); in other words reversed
   reg7 = xor(reg(8),reg8);
   reg6 = xor(reg(7),reg8);
   %shift bits left one
   reg = [reg(2:6) reg6 reg7 reg8];
end

test = [ 1 0 0 0 1 1 1 0 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1 1 0 0 1 1 0 0 ...
   0 0 0 1 1 0 0 0 0 0 1 1 0 0 1 0 zeros(1,32)];

test2 = [ 1 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 1 1 1 1 ...
   0 0 0 0 1 1 1 1];
```

```
function [output] = deCode16(bits,mod_number)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% deCode16.m
%
%  de-encode binary convolution coding; constraint length 7
%  G0 = 171; G1 = 133; uses soft decision decoding with viterbi algorithm
%
%   Input:                                                              %
%      bits:              deinterleaved bits
%      mod_number:        type of modulation
%
%   Output:                                                             %
%     output:             decoded bits
%
%   Calls:                None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

trellis = poly2trellis(7,[171 133]);
table_length = 7;

if mod_number == 0
   %To prepare for soft-decision decoding, map to decision values.
   [x,qcode] = quantiz(1-2*bits,[-.75 -.5 -.25 0 .25 .5 .75],...
   [7 6 5 4 3 2 1 0]); % Values in qcode are between 0 and 2^3-1.
   output = vitdec(qcode,trellis,table_length, 'trunc','soft',3);
elseif mod_number ==1 || mod_number ==3
   %To prepare for soft-decision decoding, map to decision values.
   [x,qcode] = quantiz(1-2*bits,[-.75 -.5 -.25 0 .25 .5 .75],...
   [7 6 5 4 3 2 1 0]); % Values in qcode are between 0 and 2^3-1.
   output = vitdec(qcode,trellis,table_length, 'trunc','soft',3,[1 1 0 1]);
elseif mod_number == 5
   %To prepare for soft-decision decoding, map to decision values.
   [x,qcode] = quantiz(1-2*bits,[-.75 -.5 -.25 0 .25 .5 .75],...
   [7 6 5 4 3 2 1 0]); % Values in qcode are between 0 and 2^3-1.
   output = vitdec(qcode,trellis,table_length, 'trunc','soft',3,[1 1 0 1 1 0]);
elseif mod_number == 2 || mod_number ==4 || mod_number == 6
   %To prepare for soft-decision decoding, map to decision values.
   [x,qcode] = quantiz(1-2*bits,[-.75 -.5 -.25 0 .25 .5 .75],...
```

```
    [7 6 5 4 3 2 1 0]); % Values in qcode are between 0 and 2^3-1.
    output = vitdec(qcode,trellis,table_length, 'trunc','soft',3,[1 1 0 1 1 0 0 1 1 0]);
end
```

```
function [output] = deinterleaver16(bits,mod_number)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% deinterleaver16.m
%
%  this function performs data interleaving per page 17 of PHY section of
% 802.11a standard
%
%   Input:                                                              %
%       bits:               vector of bits to be interleaved
%       mod_number:         type of modulation
%
%   Output:                                                             %
%       output:             decoded bits
%
%   Calls:                  None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if mod_number == 0
    first =bits([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...
        22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,...
        43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,...
        64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,...
        85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,...
        104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,...
        120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,...
        136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,...
        152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,...
        168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,...
        184,185,186,187,188,189,190,191,192]);
    output = first([1,17,33,49,65,81,97,113,129,145,161,177,2,18,34,50,...
        66,82,98,114,130,146,162,178,3,19,35,51,67,83,99,115,131,147,163,...
        179,4,20,36,52,68,84,100,116,132,148,164,180,5,21,37,53,69,85,...
        101,117,133,149,165,181,6,22,38,54,70,86,102,118,134,150,166,...
        182,7,23,39,55,71,87,103,119,135,151,167,183,8,24,40,56,72,88,...
        104,120,136,152,168,184,9,25,41,57,73,89,105,121,137,153,169,...
        185,10,26,42,58,74,90,106,122,138,154,170,186,11,27,43,59,75,91,...
        107,123,139,155,171,187,12,28,44,60,76,92,108,124,140,156,172,...
        188,13,29,45,61,77,93,109,125,141,157,173,189,14,30,46,62,78,94,...
```

```
        110,126,142,158,174,190,15,31,47,63,79,95,111,127,143,159,175,...
        191,16,32,48,64,80,96,112,128,144,160,176,192]);
elseif mod_number == 1 || mod_number == 2
    first = bits([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...
        22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,...
        43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,...
        64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,...
        85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,...
        104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,...
        120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,...
        136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,...
        152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,...
        168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,...
        184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,...
        200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,...
        216,217,218,219,220,221,222,223,224,225,226,227,228,229,230,231,...
        232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,...
        248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,...
        264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,...
        280,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,...
        296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,...
        312,313,314,315,316,317,318,319,320,321,322,323,324,325,326,327,...
        328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,...
        344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,...
        360,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,...
        376,377,378,379,380,381,382,383,384]);
    output = first([1,33,65,97,129,161,193,225,257,289,321,353,2,34,66,...
        98,130,162,194,226,258,290,322,354,3,35,67,99,131,163,195,227,...
        259,291,323,355,4,36,68,100,132,164,196,228,260,292,324,356,5,...
        37,69,101,133,165,197,229,261,293,325,357,6,38,70,102,134,166,...
        198,230,262,294,326,358,7,39,71,103,135,167,199,231,263,295,327,...
        359,8,40,72,104,136,168,200,232,264,296,328,360,9,41,73,105,137,...
        169,201,233,265,297,329,361,10,42,74,106,138,170,202,234,266,...
        298,330,362,11,43,75,107,139,171,203,235,267,299,331,363,12,44,...
        76,108,140,172,204,236,268,300,332,364,13,45,77,109,141,173,205,...
        237,269,301,333,365,14,46,78,110,142,174,206,238,270,302,334,...
        366,15,47,79,111,143,175,207,239,271,303,335,367,16,48,80,112,...
        144,176,208,240,272,304,336,368,17,49,81,113,145,177,209,241,...
        273,305,337,369,18,50,82,114,146,178,210,242,274,306,338,370,...
        19,51,83,115,147,179,211,243,275,307,339,371,20,52,84,116,148,...
        180,212,244,276,308,340,372,21,53,85,117,149,181,213,245,277,...
        309,341,373,22,54,86,118,150,182,214,246,278,310,342,374,23,55,...
        87,119,151,183,215,247,279,311,343,375,24,56,88,120,152,184,216,...
        248,280,312,344,376,25,57,89,121,153,185,217,249,281,313,345,...
        377,26,58,90,122,154,186,218,250,282,314,346,378,27,59,91,123,...
```

```
        155,187,219,251,283,315,347,379,28,60,92,124,156,188,220,252,...
        284,316,348,380,29,61,93,125,157,189,221,253,285,317,349,381,30,...
        62,94,126,158,190,222,254,286,318,350,382,31,63,95,127,159,191,...
        223,255,287,319,351,383,32,64,96,128,160,192,224,256,288,320,...
        352,384]);
elseif mod_number ==3 || mod_number ==4
    first = bits([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...
        22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,...
        43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,...
        64,66,65,68,67,70,69,72,71,74,73,76,75,78,77,80,79,82,81,84,83,...
        86,85,88,87,90,89,92,91,94,93,96,95,98,97,100,99,102,101,104,...
        103,106,105,108,107,110,109,112,111,114,113,116,115,118,117,120,...
        119,122,121,124,123,126,125,128,127,129,130,131,132,133,134,135,...
        136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,...
        152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,...
        168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,...
        184,185,186,187,188,189,190,191,192,194,193,196,195,198,197,200,...
        199,202,201,204,203,206,205,208,207,210,209,212,211,214,213,216,...
        215,218,217,220,219,222,221,224,223,226,225,228,227,230,229,232,...
        231,234,233,236,235,238,237,240,239,242,241,244,243,246,245,248,...
        247,250,249,252,251,254,253,256,255,257,258,259,260,261,262,263,...
        264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,...
        280,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,...
        296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,...
        312,313,314,315,316,317,318,319,320,322,321,324,323,326,325,328,...
        327,330,329,332,331,334,333,336,335,338,337,340,339,342,341,344,...
        343,346,345,348,347,350,349,352,351,354,353,356,355,358,357,360,...
        359,362,361,364,363,366,365,368,367,370,369,372,371,374,373,376,...
        375,378,377,380,379,382,381,384,383,385,386,387,388,389,390,391,...
        392,393,394,395,396,397,398,399,400,401,402,403,404,405,406,407,...
        408,409,410,411,412,413,414,415,416,417,418,419,420,421,422,423,...
        424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,...
        440,441,442,443,444,445,446,447,448,450,449,452,451,454,453,456,...
        455,458,457,460,459,462,461,464,463,466,465,468,467,470,469,472,...
        471,474,473,476,475,478,477,480,479,482,481,484,483,486,485,488,...
        487,490,489,492,491,494,493,496,495,498,497,500,499,502,501,504,...
        503,506,505,508,507,510,509,512,511,513,514,515,516,517,518,519,...
        520,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,...
        536,537,538,539,540,541,542,543,544,545,546,547,548,549,550,551,...
        552,553,554,555,556,557,558,559,560,561,562,563,564,565,566,567,...
        568,569,570,571,572,573,574,575,576,578,577,580,579,582,581,584,...
        583,586,585,588,587,590,589,592,591,594,593,596,595,598,597,600,...
        599,602,601,604,603,606,605,608,607,610,609,612,611,614,613,616,...
        615,618,617,620,619,622,621,624,623,626,625,628,627,630,629,632,...
        631,634,633,636,635,638,637,640,639,641,642,643,644,645,646,647,...
```

```
    648,649,650,651,652,653,654,655,656,657,658,659,660,661,662,663,...
    664,665,666,667,668,669,670,671,672,673,674,675,676,677,678,679,...
    680,681,682,683,684,685,686,687,688,689,690,691,692,693,694,695,...
    696,697,698,699,700,701,702,703,704,706,705,708,707,710,709,712,...
    711,714,713,716,715,718,717,720,719,722,721,724,723,726,725,728,...
    727,730,729,732,731,734,733,736,735,738,737,740,739,742,741,744,...
    743,746,745,748,747,750,749,752,751,754,753,756,755,758,757,760,...
    759,762,761,764,763,766,765,768,767]);
output= first([1,65,129,193,257,321,385,449,513,577,641,705,2,66,...
    130,194,258,322,386,450,514,578,642,706,3,67,131,195,259,323,...
    387,451,515,579,643,707,4,68,132,196,260,324,388,452,516,580,...
    644,708,5,69,133,197,261,325,389,453,517,581,645,709,6,70,134,...
    198,262,326,390,454,518,582,646,710,7,71,135,199,263,327,391,...
    455,519,583,647,711,8,72,136,200,264,328,392,456,520,584,648,...
    712,9,73,137,201,265,329,393,457,521,585,649,713,10,74,138,202,...
    266,330,394,458,522,586,650,714,11,75,139,203,267,331,395,459,...
    523,587,651,715,12,76,140,204,268,332,396,460,524,588,652,716,...
    13,77,141,205,269,333,397,461,525,589,653,717,14,78,142,206,270,...
    334,398,462,526,590,654,718,15,79,143,207,271,335,399,463,527,...
    591,655,719,16,80,144,208,272,336,400,464,528,592,656,720,17,81,...
    145,209,273,337,401,465,529,593,657,721,18,82,146,210,274,338,...
    402,466,530,594,658,722,19,83,147,211,275,339,403,467,531,595,...
    659,723,20,84,148,212,276,340,404,468,532,596,660,724,21,85,149,...
    213,277,341,405,469,533,597,661,725,22,86,150,214,278,342,406,...
    470,534,598,662,726,23,87,151,215,279,343,407,471,535,599,663,...
    727,24,88,152,216,280,344,408,472,536,600,664,728,25,89,153,217,...
    281,345,409,473,537,601,665,729,26,90,154,218,282,346,410,474,...
    538,602,666,730,27,91,155,219,283,347,411,475,539,603,667,731,...
    28,92,156,220,284,348,412,476,540,604,668,732,29,93,157,221,285,...
    349,413,477,541,605,669,733,30,94,158,222,286,350,414,478,542,...
    606,670,734,31,95,159,223,287,351,415,479,543,607,671,735,32,96,...
    160,224,288,352,416,480,544,608,672,736,33,97,161,225,289,353,...
    417,481,545,609,673,737,34,98,162,226,290,354,418,482,546,610,...
    674,738,35,99,163,227,291,355,419,483,547,611,675,739,36,100,...
    164,228,292,356,420,484,548,612,676,740,37,101,165,229,293,357,...
    421,485,549,613,677,741,38,102,166,230,294,358,422,486,550,614,...
    678,742,39,103,167,231,295,359,423,487,551,615,679,743,40,104,...
    168,232,296,360,424,488,552,616,680,744,41,105,169,233,297,361,...
    425,489,553,617,681,745,42,106,170,234,298,362,426,490,554,618,...
    682,746,43,107,171,235,299,363,427,491,555,619,683,747,44,108,...
    172,236,300,364,428,492,556,620,684,748,45,109,173,237,301,365,...
    429,493,557,621,685,749,46,110,174,238,302,366,430,494,558,622,...
    686,750,47,111,175,239,303,367,431,495,559,623,687,751,48,112,...
    176,240,304,368,432,496,560,624,688,752,49,113,177,241,305,369,...
    433,497,561,625,689,753,50,114,178,242,306,370,434,498,562,626,...
```

```
        690,754,51,115,179,243,307,371,435,499,563,627,691,755,52,116,...
        180,244,308,372,436,500,564,628,692,756,53,117,181,245,309,373,...
        437,501,565,629,693,757,54,118,182,246,310,374,438,502,566,630,...
        694,758,55,119,183,247,311,375,439,503,567,631,695,759,56,120,...
        184,248,312,376,440,504,568,632,696,760,57,121,185,249,313,377,...
        441,505,569,633,697,761,58,122,186,250,314,378,442,506,570,634,...
        698,762,59,123,187,251,315,379,443,507,571,635,699,763,60,124,...
        188,252,316,380,444,508,572,636,700,764,61,125,189,253,317,381,...
        445,509,573,637,701,765,62,126,190,254,318,382,446,510,574,638,...
        702,766,63,127,191,255,319,383,447,511,575,639,703,767,64,128,...
        192,256,320,384,448,512,576,640,704,768]);
elseif mod_number == 5 || mod_number ==6
    first = bits([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,...
        22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,...
        43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,...
        64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,...
        85,86,87,88,89,90,91,92,93,94,95,96,99,97,98,102,100,101,105,...
        103,104,108,106,107,111,109,110,114,112,113,117,115,116,120,118,...
        119,123,121,122,126,124,125,129,127,128,132,130,131,135,133,134,...
        138,136,137,141,139,140,144,142,143,147,145,146,150,148,149,153,...
        151,152,156,154,155,159,157,158,162,160,161,165,163,164,168,166,...
        167,171,169,170,174,172,173,177,175,176,180,178,179,183,181,182,...
        186,184,185,189,187,188,192,190,191,194,195,193,197,198,196,200,...
        201,199,203,204,202,206,207,205,209,210,208,212,213,211,215,216,...
        214,218,219,217,221,222,220,224,225,223,227,228,226,230,231,229,...
        233,234,232,236,237,235,239,240,238,242,243,241,245,246,244,248,...
        249,247,251,252,250,254,255,253,257,258,256,260,261,259,263,264,...
        262,266,267,265,269,270,268,272,273,271,275,276,274,278,279,277,...
        281,282,280,284,285,283,287,288,286,289,290,291,292,293,294,295,...
        296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,...
        312,313,314,315,316,317,318,319,320,321,322,323,324,325,326,327,...
        328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,...
        344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,...
        360,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,...
        376,377,378,379,380,381,382,383,384,387,385,386,390,388,389,393,...
        391,392,396,394,395,399,397,398,402,400,401,405,403,404,408,406,...
        407,411,409,410,414,412,413,417,415,416,420,418,419,423,421,422,...
        426,424,425,429,427,428,432,430,431,435,433,434,438,436,437,441,...
        439,440,444,442,443,447,445,446,450,448,449,453,451,452,456,454,...
        455,459,457,458,462,460,461,465,463,464,468,466,467,471,469,470,...
        474,472,473,477,475,476,480,478,479,482,483,481,485,486,484,488,...
        489,487,491,492,490,494,495,493,497,498,496,500,501,499,503,504,...
        502,506,507,505,509,510,508,512,513,511,515,516,514,518,519,517,...
        521,522,520,524,525,523,527,528,526,530,531,529,533,534,532,536,...
        537,535,539,540,538,542,543,541,545,546,544,548,549,547,551,552,...
```

```
    550,554,555,553,557,558,556,560,561,559,563,564,562,566,567,565,...
    569,570,568,572,573,571,575,576,574,577,578,579,580,581,582,583,...
    584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,...
    600,601,602,603,604,605,606,607,608,609,610,611,612,613,614,615,...
    616,617,618,619,620,621,622,623,624,625,626,627,628,629,630,631,...
    632,633,634,635,636,637,638,639,640,641,642,643,644,645,646,647,...
    648,649,650,651,652,653,654,655,656,657,658,659,660,661,662,663,...
    664,665,666,667,668,669,670,671,672,675,673,674,678,676,677,681,...
    679,680,684,682,683,687,685,686,690,688,689,693,691,692,696,694,...
    695,699,697,698,702,700,701,705,703,704,708,706,707,711,709,710,...
    714,712,713,717,715,716,720,718,719,723,721,722,726,724,725,729,...
    727,728,732,730,731,735,733,734,738,736,737,741,739,740,744,742,...
    743,747,745,746,750,748,749,753,751,752,756,754,755,759,757,758,...
    762,760,761,765,763,764,768,766,767,770,771,769,773,774,772,776,...
    777,775,779,780,778,782,783,781,785,786,784,788,789,787,791,792,...
    790,794,795,793,797,798,796,800,801,799,803,804,802,806,807,805,...
    809,810,808,812,813,811,815,816,814,818,819,817,821,822,820,824,...
    825,823,827,828,826,830,831,829,833,834,832,836,837,835,839,840,...
    838,842,843,841,845,846,844,848,849,847,851,852,850,854,855,853,...
    857,858,856,860,861,859,863,864,862,865,866,867,868,869,870,871,...
    872,873,874,875,876,877,878,879,880,881,882,883,884,885,886,887,...
    888,889,890,891,892,893,894,895,896,897,898,899,900,901,902,903,...
    904,905,906,907,908,909,910,911,912,913,914,915,916,917,918,919,...
    920,921,922,923,924,925,926,927,928,929,930,931,932,933,934,935,...
    936,937,938,939,940,941,942,943,944,945,946,947,948,949,950,951,...
    952,953,954,955,956,957,958,959,960,963,961,962,966,964,965,969,...
    967,968,972,970,971,975,973,974,978,976,977,981,979,980,984,982,...
    983,987,985,986,990,988,989,993,991,992,996,994,995,999,997,998,...
    1002,1000,1001,1005,1003,1004,1008,1006,1007,1011,1009,1010,...
    1014,1012,1013,1017,1015,1016,1020,1018,1019,1023,1021,1022,...
    1026,1024,1025,1029,1027,1028,1032,1030,1031,1035,1033,1034,...
    1038,1036,1037,1041,1039,1040,1044,1042,1043,1047,1045,1046,...
    1050,1048,1049,1053,1051,1052,1056,1054,1055,1058,1059,1057,...
    1061,1062,1060,1064,1065,1063,1067,1068,1066,1070,1071,1069,...
    1073,1074,1072,1076,1077,1075,1079,1080,1078,1082,1083,1081,...
    1085,1086,1084,1088,1089,1087,1091,1092,1090,1094,1095,1093,...
    1097,1098,1096,1100,1101,1099,1103,1104,1102,1106,1107,1105,...
    1109,1110,1108,1112,1113,1111,1115,1116,1114,1118,1119,1117,...
    1121,1122,1120,1124,1125,1123,1127,1128,1126,1130,1131,1129,...
    1133,1134,1132,1136,1137,1135,1139,1140,1138,1142,1143,1141,...
    1145,1146,1144,1148,1149,1147,1151,1152,1150]);
output = first([1,97,193,289,385,481,577,673,769,865,961,1057,2,98,...
    194,290,386,482,578,674,770,866,962,1058,3,99,195,291,387,483,...
    579,675,771,867,963,1059,4,100,196,292,388,484,580,676,772,868,...
    964,1060,5,101,197,293,389,485,581,677,773,869,965,1061,6,102,...
```

198,294,390,486,582,678,774,870,966,1062,7,103,199,295,391,487,...
583,679,775,871,967,1063,8,104,200,296,392,488,584,680,776,872,...
968,1064,9,105,201,297,393,489,585,681,777,873,969,1065,10,106,...
202,298,394,490,586,682,778,874,970,1066,11,107,203,299,395,491,...
587,683,779,875,971,1067,12,108,204,300,396,492,588,684,780,876,...
972,1068,13,109,205,301,397,493,589,685,781,877,973,1069,14,110,...
206,302,398,494,590,686,782,878,974,1070,15,111,207,303,399,495,...
591,687,783,879,975,1071,16,112,208,304,400,496,592,688,784,880,...
976,1072,17,113,209,305,401,497,593,689,785,881,977,1073,18,114,...
210,306,402,498,594,690,786,882,978,1074,19,115,211,307,403,499,...
595,691,787,883,979,1075,20,116,212,308,404,500,596,692,788,884,...
980,1076,21,117,213,309,405,501,597,693,789,885,981,1077,22,118,...
214,310,406,502,598,694,790,886,982,1078,23,119,215,311,407,503,...
599,695,791,887,983,1079,24,120,216,312,408,504,600,696,792,888,...
984,1080,25,121,217,313,409,505,601,697,793,889,985,1081,26,122,...
218,314,410,506,602,698,794,890,986,1082,27,123,219,315,411,507,...
603,699,795,891,987,1083,28,124,220,316,412,508,604,700,796,892,...
988,1084,29,125,221,317,413,509,605,701,797,893,989,1085,30,126,...
222,318,414,510,606,702,798,894,990,1086,31,127,223,319,415,511,...
607,703,799,895,991,1087,32,128,224,320,416,512,608,704,800,896,...
992,1088,33,129,225,321,417,513,609,705,801,897,993,1089,34,130,...
226,322,418,514,610,706,802,898,994,1090,35,131,227,323,419,515,...
611,707,803,899,995,1091,36,132,228,324,420,516,612,708,804,900,...
996,1092,37,133,229,325,421,517,613,709,805,901,997,1093,38,134,...
230,326,422,518,614,710,806,902,998,1094,39,135,231,327,423,519,...
615,711,807,903,999,1095,40,136,232,328,424,520,616,712,808,904,...
1000,1096,41,137,233,329,425,521,617,713,809,905,1001,1097,42,...
138,234,330,426,522,618,714,810,906,1002,1098,43,139,235,331,...
427,523,619,715,811,907,1003,1099,44,140,236,332,428,524,620,...
716,812,908,1004,1100,45,141,237,333,429,525,621,717,813,909,...
1005,1101,46,142,238,334,430,526,622,718,814,910,1006,1102,47,...
143,239,335,431,527,623,719,815,911,1007,1103,48,144,240,336,...
432,528,624,720,816,912,1008,1104,49,145,241,337,433,529,625,...
721,817,913,1009,1105,50,146,242,338,434,530,626,722,818,914,...
1010,1106,51,147,243,339,435,531,627,723,819,915,1011,1107,52,...
148,244,340,436,532,628,724,820,916,1012,1108,53,149,245,341,...
437,533,629,725,821,917,1013,1109,54,150,246,342,438,534,630,...
726,822,918,1014,1110,55,151,247,343,439,535,631,727,823,919,...
1015,1111,56,152,248,344,440,536,632,728,824,920,1016,1112,57,...
153,249,345,441,537,633,729,825,921,1017,1113,58,154,250,346,...
442,538,634,730,826,922,1018,1114,59,155,251,347,443,539,635,...
731,827,923,1019,1115,60,156,252,348,444,540,636,732,828,924,...
1020,1116,61,157,253,349,445,541,637,733,829,925,1021,1117,62,...
158,254,350,446,542,638,734,830,926,1022,1118,63,159,255,351,...
447,543,639,735,831,927,1023,1119,64,160,256,352,448,544,640,...

```
    736,832,928,1024,1120,65,161,257,353,449,545,641,737,833,929,...
    1025,1121,66,162,258,354,450,546,642,738,834,930,1026,1122,67,...
    163,259,355,451,547,643,739,835,931,1027,1123,68,164,260,356,...
    452,548,644,740,836,932,1028,1124,69,165,261,357,453,549,645,...
    741,837,933,1029,1125,70,166,262,358,454,550,646,742,838,934,...
    1030,1126,71,167,263,359,455,551,647,743,839,935,1031,1127,72,...
    168,264,360,456,552,648,744,840,936,1032,1128,73,169,265,361,...
    457,553,649,745,841,937,1033,1129,74,170,266,362,458,554,650,...
    746,842,938,1034,1130,75,171,267,363,459,555,651,747,843,939,...
    1035,1131,76,172,268,364,460,556,652,748,844,940,1036,1132,77,...
    173,269,365,461,557,653,749,845,941,1037,1133,78,174,270,366,...
    462,558,654,750,846,942,1038,1134,79,175,271,367,463,559,655,...
    751,847,943,1039,1135,80,176,272,368,464,560,656,752,848,944,...
    1040,1136,81,177,273,369,465,561,657,753,849,945,1041,1137,82,...
    178,274,370,466,562,658,754,850,946,1042,1138,83,179,275,371,...
    467,563,659,755,851,947,1043,1139,84,180,276,372,468,564,660,...
    756,852,948,1044,1140,85,181,277,373,469,565,661,757,853,949,...
    1045,1141,86,182,278,374,470,566,662,758,854,950,1046,1142,87,...
    183,279,375,471,567,663,759,855,951,1047,1143,88,184,280,376,...
    472,568,664,760,856,952,1048,1144,89,185,281,377,473,569,665,...
    761,857,953,1049,1145,90,186,282,378,474,570,666,762,858,954,...
    1050,1146,91,187,283,379,475,571,667,763,859,955,1051,1147,92,...
    188,284,380,476,572,668,764,860,956,1052,1148,93,189,285,381,...
    477,573,669,765,861,957,1053,1149,94,190,286,382,478,574,670,...
    766,862,958,1054,1150,95,191,287,383,479,575,671,767,863,959,...
    1055,1151,96,192,288,384,480,576,672,768,864,960,1056,1152]);
end
```

```
function [output,rec_fch,HCS_check] = ...
   demodData16(input,H,start_of_data,mod_number_sent)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% demodData16.m
%
%  demodulate the received signal samples after synchronization
%  perform serial to parallel conversion, take FFT, apply channel
%  correction; read PLCP header
%
%   Input:                                                              %
%      input:            received signal from channel
%      H:                channel correction values
%      start_of_data:    sample number where header starts
%      mod_number_sent:  for error checking
%
%   Output:                                                             %
%      output:           received data bits
%      rec_fch:          received header bits
%      HCS_check:        1 - good CRC check; 0 - bad CRC check
%
%   Calls:
%      FFTmap16.m:           rearrange input for FFT
%      signalDemodulate16.m: map back to bits
%      deCode16.m:           deconvolutional code
%      deRandomizer.m:       un-randomize data
%      HCS.m:                check CRC
%      deRS.m:               undo Reed-Solomon coding
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%load in stored values
load INTL;load RS; load CC; load Input_Data; load RD;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%

input = input(start_of_data:end);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%check that received data fits in blocks of 320
```

```
%based on CP of 64; 64+256 = 320
%will need to make this variable
CPSYM=320;
len = length(input);
if rem(len,CPSYM) ~= 0
   pad = CPSYM - rem(len,CPSYM);
else pad =0;
end
y1 = [input; zeros(pad,1)];
%Reshape data; 64 + 16 = 80; serial to parallel
y2=reshape(y1,CPSYM,length(y1)/CPSYM);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

y3 = y2(65:320,:);        %strip off cyclic prefix
Y=fft(y3,256);            %bring in to freq domain
%figure;plot(Y(:,1).*conj(Y(:,1)));title('Signal Spectrum')
rec_data=Y([2:13,15:38,40:63,65:88,90:101,157:168,170:193,195:218,...
   220:243,245:256],:); %remove pilots and nulls

% %plot constellation
% yplot=rec_data(:);
% h=scatterplot(yplot(1:192),1,0,'+r');
% hold on;
% scatterplot(yplot(193:end),1,0,'ob',h);
% title('Constellation of Received Signal and Data fields');
% legend('Signal','Data');
% hold off
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% %apply channel estimation correction
% H1=H([2:13,15:38,40:63,65:88,90:101,157:168,170:193,...
%   195:218,220:243,245:256]);
% corrected_data=zeros(size(rec_data));
% for i = 1:1%size(rec_data,2)
%     corrected_data(:,i) = rec_data(:,i)./H.';
% end
% ynewplot = corrected_data(:);
% scatterplot(ynewplot);title('Channel Corrected Constellation')
% rec_data = corrected_data;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Process FCH Symbol
fch = FFTmap16(rec_data(:,1));
demod_fch = signalDemodulate16(fch,0);
deint_fch = deinterleaver16(demod_fch,0);
```

```
decc_fch = deCode16(deint_fch,0);   %just cc no rs with FCH
rec_fch = deRandomizer(decc_fch,2);
rec_fch=rec_fch(1:96-8)';  %remove pad zeros

mod_number = rec_fch(17:20)'; %these bits hold rate_ID for burst
rec_hcs(1,:) = rec_fch(81:88);
calc_hcs = HCS(rec_fch(1:80));
if rec_hcs == calc_hcs
   %display('HCS Good')
   HCS_check = 1;
else
   %display('HCS Bad')
   HCS_check =0;
end

mod_number = bin2dec(num2str(mod_number));
if mod_number ~= mod_number_sent
   %display('SNR too low. Demodulation failed.')
   output =0;
   return
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Process received signal
rec_data2=rec_data(:,2:end); %strip off fch symbol

output=[];%need to need mod_num and get num of uncoded bits - 8 for rows
for i = 1: size(rec_data2,2)
   %i
   data = FFTmap16(rec_data2(:,i));
   %demodulate signal
   demod_data= signalDemodulate16(data,mod_number);
   %deinterleave
   deinterleaved_data= deinterleaver16(demod_data,mod_number);
   %de-convolution code siganl
   de_cc_data= deCode16(deinterleaved_data,mod_number);
   %de Reed Solomon code the signal
   de_rs_data = deRS(de_cc_data,mod_number);
   %derandomize the signal
   output(:,i)= deRandomizer(de_rs_data,2);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%
output = output(:);
```

```
function [output] = deRandomizer(input,burst_number)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% deRandomizer.m
%
%  this function derandomizes the data per the 802.16e standard
%  this is for the downlink only
%  input data is to enter MSB first.

%
%   Input:                                                              %
%      input:            decoded bits
%      burst_number:       which burst in transmission; for initial state
%
%   Output:                                                             %
%      output:           data bits
%
%   Calls:                None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%for testing
BSID = [0 0 0 1];
UIUC = [0 1 1 1];
Frame_number = [0 0 0 1];

%if start of frame set initial state
if burst_number ==1
   reg = [1 0 0 1 0 1 0 1 0 0 0 0 0 0 0]';
else
   reg = [BSID 1 1 UIUC 1 Frame_number]';
end

output=zeros(1,length(input));
for i = 1:length(input)
   new = xor(reg(14),reg(15));
   %shift bits left one
   reg = [new; reg(1:14)];
   % modulo 2 add output with data bit
   output(i) = xor(new,input(i));
```

```
end
```

```
function [output] = deRS(input,mod_number)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% deRS.m
%
%  this function removes the Reed-Solomon coding

%
%   Input:                                                              %
%      input:             decoded bits from binary convolution coding
%      mod_number:        type of modulation
%
%   Output:                                                             %
%      output:            bits ready for derandomizing
%
%   Calls:                None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


if mod_number ==0
   output = input(1:end-8);
   return
elseif mod_number == 1
   lim = 24;
   t=4;
elseif mod_number == 2
   lim = 36;
   t=2;
elseif mod_number == 3
   lim = 48;
   t=8;
elseif mod_number == 4
   lim = 72;
   t=4;
elseif mod_number == 5
   lim = 96;
   t=6;
elseif mod_number == 6
   lim = 108;
```

```
   t=6;
end

input=make8Bit(input);  %input to Galois field must be integer 1-2^m-1;m=8;
msg = gf(input,8);           %Create a Galois array in GF(2^8).
code = [msg(2*t+1:end) msg(1:2*t)];
code1 = [code zeros(1,255 - length(code))];
gen=rsgenpoly(255,239,285,0);
decode = rsdec(code1,255,239,gen);
decode1 = decode.x;


decode2 = decode1(1:lim); %remove redundant and zeros and zero byte tail

rs_bin_8 = zeros(1,8);
rs_out=[];
for i = 1:lim
   rs_char = dec2bin(decode2(i),8);
      for j=1:8
         rs_bin_8(j) =str2double(rs_char(j));
         rs_out = [rs_out rs_bin_8(j)];
      end
end

output=rs_out';
output=output(1:end-8);
```

```
function [signal_detected,H,start_of_data] = synch16(input,Th)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% synch16.m
%
%  this function performs signal detection using cross correlation;
%  then finds start of data and finds channel estimation
%
%   Input:                                                              %
%      input:            received samples from channel
%      Th:               detection threshold
%
%   Output:                                                             %
%      signal_detected:    1 - signal detected; 0 - no detection
%      H:                channel correction coefficients
%      start_of_data:      index for start of header
%
%   Calls:                None
%
%   Author:     Keith Howland
%   Created:    25 Mar 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


load preamble_1_fc     %time domain of first 64 samples
                       %of first preamble symbol, flipped and
                       %conjugated for correlation
load long_preamble
signal_detected=0;
%first preamble correlation
lp_power = long_preamble(1:64)'*long_preamble(1:64);
Rsp = filter(preamble_1_fc,1,input)/lp_power;
mRsp=(Rsp .* conj(Rsp));
%figure;plot(mRsp);title('Magnitude of first Preamble Autocorrelation')
%xlabel('Sample Index n');ylabel('Magnitude');

%64 samples after last peak starts the long preamble
for i = 1:length(mRsp)
   if mRsp(i) > Th && mRsp(i-64) > Th && mRsp(i-64*2) > Th ...
         && mRsp(i-64*3) > Th && mRsp(i-64*4) > Th && mRsp(i-64*5) < Th
      start_of_data = i + 320 + 1;
      signal_detected=1;
```

```
    end
end

if signal_detected ==0  %if signal not detected
    H=0;start_of_data=0;
    return
end


%Estimate Channel
% L=16;                    %length of channel set by standard to 15
%
% load /Receiver/preamble_2_128
% X = fft(preamble_2_128);
% X1 = [X(1:50); X(52:101)];
%
% LP=input(start_of_data-256:start_of_data-1);
% lp1=LP(1:128);              %1st Long Preamble
% lp2=LP(129:256); %2nd Long Preabmle
% LP1 = fft(lp1);
% LP2 = fft(lp2);
% Y1 = [LP1(1:50); LP1(52:101)];
% Y2 = [LP2(1:50); LP2(52:101)];
% H = (Y1+ Y2)./(2*X1);
% H2(1:2:200) = H;
% H2(2:2:200) = H;
% H3 = H2(1:192);
% figure;plot(abs(H3));
% h = ifft(H3,256);
% figure;plot(abs(h));title('Channel impulse response')
% H = H3;

% xLP=preamble_2_128;  % Long Preamble
% L=64;  % choose L>16 to account for uncertainty in the estimated beginning of the
preamble
% X=toeplitz(xLP, [xLP(1); xLP(128:-1:128-L+2)]);
% Xinv=inv(X'*X)*X';
% h1=Xinv*LP1;figure;plot(abs(h1));
% h2=Xinv*LP2;figure;plot(abs(h2));
% h=0.5*(h1+h2);
% H=fft(h,256);
% figure;plot(abs(H));
H=ones(256,1);
```

```
function [output,sent_plcp,MAC_data] = build80211b(MAC_bytes)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% build80211b.m
%
%  make the IEEE 802.11b signal; only uses differential BPSK modulation
%
%   Input:                                                              %
%      input:
%          MAC_data:   data bits from MAC layer
%          MAC_bytes:  number of bytes of data from MAC layer
%
%   Output:                                                             %
%      output:     modulated signal
%      sent_plcp:  sent plcp header bits for BER check
%
%   Calls:
%      LongPLCP.m to Preamble and Header
%
%   Author:     Keith Howland
%   Created:    30 May 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
MAC_data = randint(1,MAC_bytes*8);
PLCP = LongPLCP(1,MAC_bytes);  %make PLCP with data rate = 1 Mbps
sent_plcp = PLCP(145:192); %PLCP Header bits: 48
PPDU = [PLCP MAC_data];  %add MAC layer data after preamble and header
%gray coded binary DBPSK; no phase offset
output = dpskmod(PPDU,2,0,'gray');
```

```
function [output] = CRC11b(input)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% LongPLCP.m
%
%   create cyclic redundancy check bits
%   G(x) = X^16 + X^12 + X^5 + 1
%
%   Input:                                                              %
%      input:      PLCP header bits
%
%   Output:                                                             %
%      output:     16 CRC bits
%
%   Calls:          None
%
%   Author:     Keith Howland
%   Created:    23 May 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

reg = ones(1,16);   %set initial state
for i = 1:length(input)
   reg16 = xor(reg(1),input(i));   %my reg16 = X(0);
   reg5 = xor(reg(5),reg16);
   reg12 = xor(reg(12),reg16);
   %shift bits left one
   reg = [reg(2:4) reg5 reg(6:11) reg12 reg(13:16) reg16];
end

output = (reg == 0);   %output is ones complement MSB sent first

%test = [0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0];
```

```
function output = LongPLCP(data_rate,num_data_octets)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% LongPLCP.m
%
%  make the IEEE 802.11b Preabmle and Header
%
%   Input:                                                              %
%      data_rate:      only using 1 Mbps
%      num_octets:     how many bytes of MAC layer data
%
%   Output:                                                             %
%      output:         Preamble and PLCP header
%
%   Calls:
%      Scrambler11b.m to make Preamble
%      CRC11b.m to create cyclic redundancy check bits
%
%   Author:     Keith Howland
%   Created:    23 May 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
x = ones(1,128);    %starting 128 bits of ones
SYNC = Scrambler11b(x);  %scramble them
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%static SFD field
%reverser order because right most transmitted first
SFD = [0 0 0 0 0 1 0 1 1 1 0 0 1 1 1 1];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%SIGNAL field
if data_rate == 1
   SIGNAL = [0 1 0 1 0 0 0 0];     %LSB first
elseif data_rate == 2
   SIGNAL = [0 0 1 0 1 0 0 0];
elseif data_rate == 5.5
   SIGNAL = [1 1 1 0 1 1 0 0];
elseif data_rate == 11
   SIGNAL = [0 1 1 1 0 1 1 0];
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%calculate LENGTH field: number of microseconds to transmit PSDU
P=0;
LENGTH_prime =((num_data_octets + P)* 8)/data_rate; %page 16 of 802.11b-99
LENGTH = ceil(LENGTH_prime);
if data_rate == 11 && LENGTH - LENGTH_prime >= 8/11
   LENGTH_EXTENSION = 1;
else
    LENGTH_EXTENSION = 0;
end
LENGTH_bin = dec2bin(LENGTH,16);
LENGTH_bits = zeros(1,16);  %preallocate space
for j = 1:16
    %convert each bit from string
    LENGTH_bits(1,j) = str2double(LENGTH_bin(j));
end
LENGTH = fliplr(LENGTH_bits);   %LSB sent first
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%
%bits 2,3 are variable but don't matter now
SERVICE = [0 0 0 0 0 0 0 LENGTH_EXTENSION];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%
CRC_input = [ SIGNAL SERVICE LENGTH];   %input for CRC
CRC_bits = CRC11b(CRC_input);          %calculate CRC
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%

output =[SYNC SFD SIGNAL SERVICE LENGTH CRC_bits];
```

```
function [output] = Scrambler11b(input)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% Scrabmler11b.m
%
%  Scramble bits to create IEEE 802.11b Preamble
%  G(z) = Z^-7 + Z^-4 + 1
%
%   Input:                                                              %
%      input:      all ones
%
%   Output:                                                             %
%      output:     scrambled bits
%
%   Author:     Keith Howland
%   Created:    23 Feb 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

output=zeros(1,length(input));  %preallocate for speed

reg = [1 1 0 1 1 0 0];  %set initial state
for i = 1:length(input)
   new = xor(reg(4),reg(7));
   %shift bits left one
   reg = [new reg([1 2 3 4 5 6])];
   % modulo 2 add output with data bit
   output(i) = xor(new,input(i));
end
```

```
function [output] = deCRC(CRC_bits, CRC_input)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% deCRC.m
%
%  calculate CRC bits to check for bit errors during transmission
%  G(x) = X^16 + X^12 + X^5 + 1
%  if received crc bis match calculated crc then good
%
%   Input:                                                              %
%      CRC_bits:       received CRC value
%      CRC_input:      received signal field for crc calculation
%
%   Output:                                                             %
%      output:         1 - good CRC check
%                      0 - bad CRC check
%
%   Calls:              none
%
%   Author:     Keith Howland
%   Created:    30 May 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


reg = ones(1,16);   %set initial state to all ones
for i = 1:length(CRC_input)
   reg16 = xor(reg(1),CRC_input(i));   %my reg16 = X(0);
   reg5 = xor(reg(5),reg16);
   reg12 = xor(reg(12),reg16);
   %shift bits left one
   reg = [reg(2:4) reg5 reg(6:11) reg12 reg(13:16) reg16];
end

%set initial state;%need to take ones complement first
sent_reg = (CRC_bits == 0);

if reg == sent_reg
   %good CRC check: received matched calculated
   output = 1;
else
   %bad CRC check
   output = 0;
```

```
end
```

```
function [data_rate,num_data_octets,rec_data,rec_header,CRC_result] = ...
   demodData11b(noise_signal,SIGNAL_start)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% demodData11b.m
%
%  Demodulate the received signal which can only be sent using DBPSK
%  then read SIGNAL field to extract header information
%
%   Input:                                                              %
%      noise_signal:       received signal from channel
%      SIGNAL_start:       start index of signal field
%
%   Output:                                                             %
%      data_rate:          received data rate from SIGNAL field
%      num_data_octets:    number of bytes of data sent from LENGTH field
%      rec_data:           demodulated data bits
%      rec_header:         received PLCP header bits for BER
%
%   Calls:
%      deCRC.m             to calculate cyclic redundancy check
%
%   Author:     Keith Howland
%   Created:    23 May 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%demodulate received signal
rec_signal = dpskdemod(noise_signal,2,0,'gray');

%remove preamble

rec_header = rec_signal(SIGNAL_start: SIGNAL_start+47)';%PLCP Header 48 bits
rec_data = rec_signal(SIGNAL_start +48:end);

SIGNAL = rec_header(1:8);    %read SIGNAL field
MOD = rec_header(12);        %type of transmission
LENGTH_EXTENSION = rec_header(16);
LENGTH = rec_header(17:32);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%check cyclic redundancy check bits
```

```
CRC_bits = rec_header(33:48);
CRC_input = rec_header(1:32);
CRC_result = deCRC(CRC_bits,CRC_input);
if CRC_result == 0
   %if bad return with all parameters equal to zero
   display('BAD CRC. Demodulation failed.')
   data_rate =0;
   num_data_octets =0;
   rec_data =0;
   return
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%find data rate
if SIGNAL == [0 1 0 1 0 0 0 0]
   data_rate = 1;
elseif SIGNAL == [0 0 1 0 1 0 0 0]
   data_rate = 2;
elseif SIGNAL == [1 1 1 0 1 1 0 0]
   data_rate = 5.5;
elseif SIGNAL == [0 1 1 1 0 1 1 0]
   data_rate = 11;
else
   %if undetected bit error by CRC
   display('Demodulation failed')
   return
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
LENGTH = fliplr(LENGTH);
len = bin2dec(num2str(LENGTH));

num_data_octets = floor(len * data_rate/8 - MOD) - LENGTH_EXTENSION;
```

```
function [signal_detected11b,start_of_SIGNAL] = synch11b(input,Th)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
% synch11b.m
%
%  Synchronize to received IEEE 802.11b signal using cross-correlation
%  witht the known preamble; determine start of data sample index
%
%   Input:                                                              %
%      input:      received signal sample values
%      Th:         decision threshold
%
%   Output:                                                             %
%      signal_deteced11b:      1 - 802.11b signal was detected
%                              0 - 802.11b signal was not detected
%      start_of_SIGNAL:        sample index of start of SIGNAL field
%
%   Calls:          None
%
%   Author:     Keith Howland
%   Created:    25 May 07
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

load preamble;  %pre-saved preamble for cross correlation
p_corr = filter(preamble,1,input);  %do the correlation
preamble_power = preamble *preamble';  %preamble power
p_corr = p_corr/ preamble_power*10;  %normalize wrt preamble power
mag_p_corr = p_corr .* conj(p_corr);    %get magnitude
%figure;plot(mag_p_corr);title('Preamble Correlation')

[v,ind1] = max(mag_p_corr);

if v > Th
   signal_detected11b = 1;
else
   %no signal detected, so set all parameters to zero
   %disp('not 802.11b')
   signal_detected11b =0;
   start_of_SIGNAL =0;
   return
end
```

```
%start of data
start_of_SFD = ind1 +1;
start_of_SIGNAL = start_of_SFD + 16;
```

LIST OF REFERENCES

[1] Institute of Electrical and Electronics Engineers, 802.11a, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: High-Speed Physical Layer Extension in the 5 GHz Band, 12 June 2003. http://ieeexplorer.ieee.org, last accessed 15 August 2007.

[2] Institute of Electrical and Electronics Engineers, 802.11b, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band, 16 September 1999. http://ieeexplorer.ieee.org, last accessed 15 August 2007.

[3] A. Ghosh, D. Walter, J. Andrews, R. Chen, "Broadband wireless access with WiMax/802.16: Current Performance Benchmarks and Future Potential," *IEEE Communications Magazine*, vol. 43, pp. 129 – 136, February 2005.

[4] T. Schmidl and D. Cox, "Robust frequency and timing synchronization for OFDM," *IEEE Trans. on Communications*, vol. 45, no. 12, pp. 1613– 1621, December 1997.

[5] J. van de Beek, M. Sandell, P. Borjesson, "ML estimation of time and frequency offset in OFDM systems," *IEEE Trans. on Signal Processing*, vol. 45, no. 7, pp. 1800-1805, July 1997.

[6] P. Moose, "A technique for orthogonal frequency division multiplexing frequency offset correction," *IEEE Trans. On Communications*, vol. 42, no. 10, pp. 2908-2914, October 1994.

[7] Larsson, G. Liu, J. Li, G. Giannakis, "Joint symbol timing and channel estimation for OFDM base WLANs," *IEEE Communications Letters*, vol. 5, no. 8, pp. 325-327, August 2001.

[8] K. Yip, T. Ng and Y. Wu, "Impacts of multipath fading on the timing synchronization of IEEE 802.11a wireless LANs," in *Proc. IEEE International Conference on Communications*, New York, New York, May 2002, pp. 517 – 521.

[9] Y. Wu, K. Yip, T. Ng, E. Serpdin, "Maximum-likelihood symbol synchronization for IEEE 802.11a WLANs in unknown frequency-selective fading channels," *IEEE Trans. on Wireless Communications*, vol. 4, no. 6, pp. 2751-2763, November 2005.

[10] T. Rappaport, Wireless Communications, pp. 197-210, Prentice Hall, 2002.

[11] J. Proakis, Digital Communications, pp. 160-163, McGraw Hill, 2001.

[12] Institute of Electrical and Electronics Engineers, 802.16, Air Interface For Fixed Broadband Wireless Access Systems, 1 October 2004. http://ieeexplorer.ieee.org, last accessed 15 August 2007.

[13] R. Cristi, Modern Digital Signal Processing, pp.111-113, Brooks/Cole, 2004.

[14] H. Van Trees, Detection, Estimation, and Modulation Theory, pp. 19-40, John Wiley and Sons, 1968.

[15] R. Hippenstiel, Detection Theory, pp. 67-96, CRC Press, 2002.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chair, Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Professor Murali Tummala
   Naval Postgraduate School
   Monterey, California

5. Professor John McEachen
   Naval Postgraduate School
   Monterey, California